National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

Search Strategies for Conspiracy Numbers

by

Norbert L. Klingbeil

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1989

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-52837-0

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Norbert L. Klingbeil

TITLE OF THESIS: Search Strategies for Conspiracy Numbers

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....

Permanent Address:
10524-154 Avenue
Edmonton, Alberta
Canada T5X 5A9

Dated 21 December 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Search Strategies for Conspiracy Numbers** submitted by **Norbert L. Klingbeil** in partial fulfillment of the requirements for the degree of **Master of Science.**

J. Schaeffer
Supervisor

T. A. Marsland

J. E. Innis

W. Bolton

Date December 21, 1988

ABSTRACT

McAllester's Conspiracy Numbers algorithm is an exciting, new minimax search procedure that builds game trees to a variable depth without application-dependent knowledge. The algorithm selectively expands nodes in the tree trying to narrow the range of likely minimax values at the root of the search tree. The likelihood of the root achieving a value is expressed as that value's conspiracy number: the minimum number of leaf nodes required to conspire, by changing their values together, to cause the root to change to that value.

The main issue involved in implementing this algorithm is a space-time trade-off. Three implementation versions were tested that demonstrate this trade-off. These versions are distinguished by the number of conspiracy numbers each stores at nodes. It is shown that storing conspiracy numbers at nodes significantly improves the algorithm's performance.

Initial experience with using the algorithm in a chess program indicated several problems with it. The main problem is that the algorithm places too much emphasis on depth rather than breadth of search. Several variations of the algorithm are demonstrated, each adding an increasing degree of breadth to the search. Experiments showed that the new algorithms are capable of solving up to 41% more chess positions than McAllister's original proposal.

Acknowledgements

I would like to thank my supervisor, Jonathan Schaeffer, for his guidance, encouragement, and constant prodding without which this thesis never would have been completed. I would also like to thank my parents, Assar and Anni Klingbeil, for their patience and support throughout this endeavor. And finally, special thanks to Robert Enns, who co-authored the three implementation versions of the Conspiracy Numbers Algorithm, to Daniel Lanovaz, who proof-read and provided valuable feedback on parts of this thesis, and to the members of my examining committee, Dr. Wlodec Dobosiewicz, Dr. Tony Marsland, and Dr. Ted Lewis, for their valuable time and suggestions.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Search Trees	2
1.2. Search Algorithms	3
1.3. Thesis Description	8
1.4. Thesis Outline	8
Chapter 2: The Conspiracy Numbers Algorithm	10
2.1. An Overview of the Algorithm	10
2.2. Computing Conspiracy Numbers	12
2.3. Computing the Range of Likely Root Values	15
2.4. Terminating the Search	16
2.5. Strategies for Narrowing the Range of Likely Root Values.	16
2.6. Selecting a Leaf Node for Expansion	17
2.7. An Example	20
2.8. Comments	22
Chapter 3: Implementing Conspiracy Numbers	25
3.1. A Basic Implementation	26
3.1.1. The Data Structures	26
3.1.2. Allowable Minimax Values	27
3.1.3. The Routines	28
3.1.4. Calling the Top Level Routine	32
3.2. A Storage Intensive Implementation	32
3.2.1. The Data Structures	33
3.2.2. Maintaining the Conspiracy Numbers	36
3.3. A Compromise Implementation	37
3.3.1. Which Conspiracy Numbers to Store	37
3.3.2. Maintaining the Conspiracy Numbers	39
3.3.3. Updating the Conspiracy Numbers when Vmin or Vmax Change	40
3.3.4. Updating the Conspiracy Numbers after an Expansion	42
3.4. Conclusion	45
Chapter 4: Results for Random Trees	47
4.1. Performance Results	47
4.2. General Results	54
4.3. Conclusion	56
Chapter 5: Incorporating Conspiracy Numbers in a Chess Program	58
5.1. Implementing Conspiracy Numbers in Tactical Chess	59
5.1.1. Integrating Chess Knowledge into the Algorithm	59
5.1.2. Basic Control Mechanism	61

5.1.3. Problems 63
5.1.4. An Enhancement 65
5.2. Experimenting with Conspiracy Numbers in Tactical Chess 65
5.2.1. The Program 66
5.2.2. The Test Data 68
5.2.3. Initial Results 68
5.2.4. Search Behavior 72
5.2.5. Adjusting Parameters 74
5.2.6. Summary of Results 75
Chapter 6: Improving the Search Strategy 76
6.1. Alternative Rules for Choosing a Branch 78
6.1.1. Rules Requiring No Additional Computed Information 79
6.1.2. Rules Requiring Additional Computed Information 82
6.2. An Alternative Definition to Convergence 87
6.3. Considering Conspiracy Set for Alternative Values 90
6.4. Ordering the New Leaf Nodes 91
6.5. Results 92
6.6. Conclusion 99
Chapter 7: Conclusions and Further Work 100
7.1. Results 100
7.2. Further Work 101
References 103
Appendix 1: The Basic Implementation Version 104
Appendix 2: The Storage Intensive Implementation Version 109
Appendix 3: The Compromise Implementation Version 114

List of Figures

Figure		Page
1.1	A Generic Search Algorithm.	4
2.1	Outline of Conspiracy Numbers Algorithm.	11
2.2	A search tree.	12
2.3	Conspirators.	13
2.4	Conspiracy Numbers for the search tree.	14
2.5	Choosing a strategy.	17
2.6	Procedures for finding leaf nodes.	19
2.7	The initial tree.	20
2.8	The tree after one expansion.	20
2.9	The tree after two expansions.	21
2.10	The tree after three expansions.	22
2.11	Alternative rule for choosing a strategy.	23
3.1	A search tree in the Basic Implementation.	27
3.2	The Top_Level Routine.	29
3.3	A search tree in the Storage Intensive Implementation.	35
4.1	Initial Performance Comparisons.	47
4.2	Profiles of the Implementations.	48
4.3	Performances for different range sizes.	50
4.4	Performances for different branching factors.	51
4.5	Performances for different average depths.	52
4.6	Storage Requirements of the three versions.	54
4.7	Number of expansions until total convergence occurs.	55
5.1	Overview of our chess program.	59
5.2	Number of problems solved.	68
5.3	Conspiracy thresholds of solved problems.	69
5.4	Final conspiracy thresholds of unsolved problems.	70
5.5	Storage requirements for search trees.	71
5.6	Distribution of nodes.	72
5.7	Results for different range sizes.	74
5.8	Results for different depth limits.	75
6.1	Results for different ordering methods.	92
6.2	Results for the round robin version.	94
6.3	Initial results for the likelihood version.	94
6.4	Results for variants of the likelihood version.	96
6.5	Results for the Best Move Cutoff enhancement.	97
6.6	Results for the Minimum Windows enhancement.	99

Chapter 1

Introduction

We are interested in two-player adversary games like chess. More specifically, we are interested in programs that play such types of games. Games appear to be a good domain in which to explore machine intelligence because they provide a structured task in which success or failure is easily measured. To further limit the type of games we are interested in, assume that each player must function with complete information about what may happen (i.e. there is no element of "chance" in the game), and that the search space is large enough so that a player can only examine a small portion of it before he has to make a move.

This type of game-playing program has one basic goal: to choose the best move to make during its turn. To do this, it performs a sequence of three actions:

1) generate all the legal moves,

2) evaluate each move,

3) choose the best move.

The most critical and difficult portion of the program is the evaluation of each move. In its simplest form, the program can examine the game states resulting from each move in order to evaluate them. However, a program that looks at a series of additional moves to predict a likely outcome will be able to play a much better game. Therefore, many programs that play games examine as many future states as time permits in order to better evaluate its available moves.

Several algorithms, such as Alpha-Beta [KnM75], were developed for systematically exploring future game states. These algorithms are often referred to as *search algorithms*. They are typically application independent and form the backbone of many game-playing programs. This thesis examines a new search algorithm called Conspiracy Numbers [McA88].

1.1. Search Trees

A search algorithm starts with an initial state and builds a search tree. Each node in the search tree corresponds to a game state. The root node corresponds to the initial state. Each node will contain several pieces of information including the *move* that must be applied to its parent in order to reach it and a *value* representing an evaluation of its corresponding state.

The *levels* in a search tree are numbered in ascending order beginning with zero. The root node is the only node that appears on level zero. Nodes on the same level all require the same number of moves to be made (starting from the root node) in order to reach them.

Nodes can be classified according to whose turn it is. For example, the root node corresponds to a game state where it is the program's turn to make a move, and its sons correspond to states where it is the opponent's turn to make a move. In fact, all nodes on even numbered levels represent states where it is the program's turn and all nodes on odd numbered levels represent states where it is the opponent's turn.

A *static evaluator function* evaluates individual game states, using whatever application-dependent information it has, by estimating the likelihood of that state eventually leading to a win. The function return values are typically from a range $[l, u]$, where the values from the upper end of the range (u) indicate that the state favors the program and the values from the lower end (l) indicate that the state favors the opponent.

The static evaluation function is only applied at leaf nodes in the search tree. States that correspond to the various leaf nodes represent possible "target" states that a player may try to "travel" (i.e. move) towards or try to avoid. States that correspond to the various interior nodes represent states that will only be passed through. At interior nodes, the program is not interested in the static evaluation of the states they

represent, but rather in their eventual outcomes (i.e. the most likely leaf state that will be reached). The value assigned to an interior node usually indicates the most likely outcome and is therefore a function of the values of the leaf nodes beneath it.

For the search algorithm that we will be studying, the node values that are stored at interior nodes are called *minimax values*. Computing the minimax values is simple. Recall that game states are evaluated from the program's perspective. At nodes where it is the program's turn to move, the move corresponding to the son with the maximum value is chosen. At nodes where it is the opponent's turn to move, the move corresponding to the son with the minimum value is chosen.[1] In other words, each player chooses the move that will lead to a game state that is most favorable to himself. The minimax value is simply the value of the corresponding chosen move. In this way, the minimax value of an interior node is identical to the value of the most likely reachable leaf state.

1.2. Search Algorithms

Figure 1.1 provides a description of a generic search algorithm. Its main task is to grow (or extend) a search tree. Assume here that a search tree was initialized beforehand (i.e. it already exists). The algorithm consists of one main loop that first checks for a completed search. If the search is not complete, a leaf node will be selected for expansion by the function *select_node()*. In selecting such a node, the algorithm forms a *search path* consisting of an ordered set of nodes from the root node to the node selected for expansion. When a node is placed in this search path, it is said to be in the process of being *explored*. In expanding a leaf node, the function *expand_node()* generates the set of moves that will be considered from the state corresponding to this node. It then creates a set of new nodes, assigns the moves to them, and evaluates their corresponding states. Finally, the new values have to be propagated up the

[1] These nodes are often referred to as a "minimizing node" or "maximizing node".

search path by the function *backup_values()*. That is, the node values at nodes along the search path must be recomputed in order to reflect the new information provided by these new nodes. If the search is finished, a move is selected by the function *best_move()*. Typically, this move corresponds to the node on level 1 with the maximum value. This move is then returned by the procedure to the game-playing program.

```
generic_search()
{
    while not done() do {
        n = select_node();
        expand_node(n);
        backup_values();
    }
    return( best_move() );
}
```

Figure 1.1 A Generic Search Algorithm.

We will now examine some actual algorithms. The focus will mainly be on the termination decision and the node selection decision because they provide most of the distinction between the algorithms.

The most basic of the algorithms is the *minimax search algorithm*. It grows a search tree in a depth-first manner to a preselected depth and computes minimax values for all interior nodes. The search is terminated when all possible nodes up to the preselected depth have been generated. If the average branching factor for a search tree is w, the computing time of this algorithm is $O(w^d)$ where d is the depth of the search. Therefore, it is impossible to grow such a tree to an arbitrary depth and a reasonable depth limit must always be applied. However, a program that can search $d + 1$ moves (called "ply" in game-playing literature) deep should play better than a program that only searches d moves deep.[2] Hence techniques for increasing the depth

[2] Except in pathological trees. See [Nau79].

achievable should be sought.

The *Alpha-Beta search algorithm* is an enhanced version of the minimax search algorithm. Since the minimax search procedure is depth-first, its efficiency can be improved by using branch-and-bound techniques. In the Alpha-Beta search procedure, exploration of a branch is abandoned early if it is proven that the move it represents is worse than a previously explored move. This results in a significant saving in search effort; in a best case scenario, the computing time of Alpha-Beta is $O(w^{d/2})$ [KnM75].

There are two problems with the Alpha-Beta algorithm. The first is that Alpha-Beta wastes a considerable amount of time exploring subtrees that have only a small chance of success, but must be considered to be certain Alpha-Beta returns the correct result. The second problem is that Alpha-Beta cannot follow promising lines of play because they are cut off whenever the depth limit is reached. However, a new algorithm for searching minimax trees elegantly solves both of these problems.

McAllester's *Conspiracy Numbers algorithm* grows a search tree in such a way that leads to a reasonable confidence in the accuracy of the root's minimax value. By the term "reasonable confidence" we mean that the root's value has become stable and that it would not change much if the tree were expanded further. By selectively expanding only certain parts of the search tree, this algorithm grows trees that are shaped in non-uniform ways.

Conspiracy numbers provide a measure of the difficulty of changing a node's value to some other value. Specifically, they measure the minimum number of leaf nodes that must change their value appropriately (by being searched deeper) to cause this node to change to that value. Conspiracy numbers at the root node, combined with a threshold value, are used to compute a range of likely root values. This range is denoted by [*Vmin*, *Vmax*] where *Vmin* is the lower bound value and *Vmax* is the upper bound value. Values outside this range are considered unlikely because their

corresponding conspiracy numbers were greater than the threshold, whereas values inside this range are considered likely root values because their conspiracy numbers were less than the threshold. This range includes the current root value and represents the "accuracy" of the root value.

The algorithm extends the search tree in such a way that quickly narrows this range. By exploring those leaf nodes that must "conspire" to change the root node to a likely root value,[3] the algorithm either succeeds in changing the root's value or succeeds in increasing that value's corresponding conspiracy number beyond the threshold. This results in greater exploration of branches that are more likely to affect the root value, especially along interesting lines of play. The algorithm terminates when the range becomes small enough (i.e. when the root value is judged to be accurate enough).[4]

The amount of confidence in the final root value is primarily a function of the threshold. Assume that the search was terminated when the current root value was the only member of the range. A higher threshold will mean greater confidence in that value because of the increased effort required to change it to another value.

Berliner's *B* Search Algorithm* [Ber79,Pal82] also selectively expands nodes in order to grow a search tree. However, it is different from the Conspiracy Numbers algorithm. The biggest difference is that in B* the value associated with each node is a range whereas in Conspiracy Numbers it is a single number. In B* the static evaluation function at leaf nodes returns two values, a pessimistic value and an optimistic value, which are propagated up the tree in order to compute interior node values. However, the B* algorithm relies on the true value of each node always being within the given range.

---

[3] Usually one of the endpoint values is chosen (i.e. Vmin or Vmax).

[4] In most applications one would like to continue the search until the current root value is the only member of this range.

The object of the B* search algorithm is to find a move at the root that is proven better than all others. In other words, the algorithm searches until it has found a move whose pessimistic value is at least as good as all other moves' optimistic value.

The B* algorithm relies on two strategies to "separate" out the best move. The *ProveBest* strategy tries to raise the pessimistic value of the best move above the optimistic values of the other moves by further exploring the sub-tree corresponding to the best move. The *DisproveRest* strategy tries to lower the optimistic value of one of the other moves to below the pessimistic value of the best move by further exploring the sub-tree corresponding to one of these other moves. The decision regarding which strategy to apply followed by the decision on which leaf nodes to expand is made solely on the basis of probabilities. These probabilities measure the likelihood of changing the bounds on the intervals and hence are a function of the amount of change desired, the probability distribution over the interval, and the size of the interval. Basically the algorithm wants to "separate" out a best move at the root by performing the least amount of work and therefore chooses to perform those strategies and to expand those leaf nodes that are most likely to succeed.

A deficiency of the B* algorithm is that it relies on the true value of each node always being within the given range, putting tremendous onus on the accuracy of the static evaluation function. If the function fails to compute an encompassing range, the algorithm may be unsuccessful. In the Conspiracy Numbers algorithm, ranges can also be computed at each node using conspiracy numbers and thresholds. In fact, one can view the algorithm as using these ranges, but in a different way than B* does. However, a major difference between these two algorithms is on what they base their ranges on. In Conspiracy Numbers, the range at a leaf node is completely open (i.e. $[-\infty,+\infty]$). It only closes when the leaf node is explored. In other words, ranges in Conspiracy Numbers are based completely on the search tree grown below a node. In B*, ranges are based on the limited knowledge inside an evaluation function.

1.3. Thesis Description

As stated earlier, this thesis examines the Conspiracy Numbers algorithm. Initial work on this algorithm started shortly after the technical report describing it became available. At that time, no known implementation of the algorithm had existed. Therefore, considerable effort was initially put into developing an efficient implementation of the algorithm and testing it in an application.

Specifically, this is a *practical* study of the Conspiracy Numbers algorithm. The objectives of this thesis are to:

- examine implementation issues of the algorithm,
- examine issues of implementing it in a practical domain, in this case the game of chess,
- understand its behavior in searching chess trees as well as randomly generated trees,
- and to test modifications to the algorithm in chess.

1.4. Thesis Outline

The Conspiracy Numbers algorithm is described in Chapter 2. This description is based on McAllester's original technical report [McA85]. A later article on the algorithm by McAllester [McA88] differs slightly in its use of the conspiracy threshold for computing *Vmin* and *Vmax*.

Chapter 3 examines the implementation of the algorithm. The main issue involved in implementing it is the space-time trade-off. To investigate this trade-off, three implementation versions are presented.

Chapter 4 compares the performances of the three implementation versions on randomly generated trees. It also examines the factors that affect their performances and studies the algorithm's behavior in searching randomly generated trees.

Chapter 5 describes how the algorithm was adapted for implementation in a program that solves tactical chess problems. Experiments compare the performance of this program to the performance of a similar program that uses the Alpha-Beta search algorithm.

Chapter 6 describes several performance enhancing modifications to the algorithm. These modifications are tested in the above chess program and the experiment results are reported.

Finally, chapter 7 presents the conclusions and further work.

Chapter 2

The Conspiracy Numbers Algorithm

This chapter presents the Conspiracy Numbers algorithm. The first section is an overview of the algorithm; it identifies the major aspects of the algorithm. Subsequent sections explain in more detail the major parts of the algorithm. An example of how a tree is grown using this algorithm along with some comments about the algorithm conclude the chapter.

2.1. An Overview of the Algorithm

The new tree-growth procedure presented here is based on a measure of the *accuracy* of the minimax value of a search tree. A root value is said to be accurate if further search is unlikely to change that value. Search trees are grown so that the minimax root values become accurate.

Figure 2.1 provides an outline of the Conspiracy Numbers algorithm. Assume that the search tree has been initialized beforehand, so that it consists of at least a root node. The algorithm takes two parameters: a range parameter Δ which states the accuracy to which the root value is to be determined, and a conspiracy threshold CT which states the degree of confidence we would like in our answer. The answer this algorithm returns is the root value ($Vroot$).

A range of likely root values, denoted by $[Vmin, Vmax]$, is used to represent the accuracy of the root value. Here $Vmin$ denotes a lower bound on the root value and $Vmax$ denotes an upper bound. Conspiracy numbers computed at the root node, combined with the conspiracy threshold CT, are used to determine $Vmin$ and $Vmax$.

The algorithm consists of a main loop that first checks whether the search has been completed. The search is terminated when the accuracy of the root value is such that $Vmax - Vmin \leq \Delta$.

```
CN_Search( Δ, CT )
{
    compute conspiracy numbers at root node
    compute Vmin, Vmax
    while ( Vmax - Vmin > Δ ) do {
        choose to rule out Vmax or Vmin
        select appropriate leaf
        expand leaf node
        compute minimax values along search path
        compute conspiracy numbers at root node
        compute Vmin, Vmax
    }
    return( Vroot )
}
```

Figure 2.1 Outline of Conspiracy Numbers Algorithm.

The algorithm extends the search tree in such a way so that it quickly narrows the range of likely root values. To do this, the algorithm relies on two strategies: *rule_out_Vmin* and *rule_out_Vmax*. Having selected a strategy, the algorithm then selects an appropriate leaf node to expand.

After a leaf node is expanded, the minimax values along the search path are recomputed. And finally, in preparation for the next iteration, *Vmin* and *Vmax* are also recomputed.

2.2. Computing Conspiracy Numbers

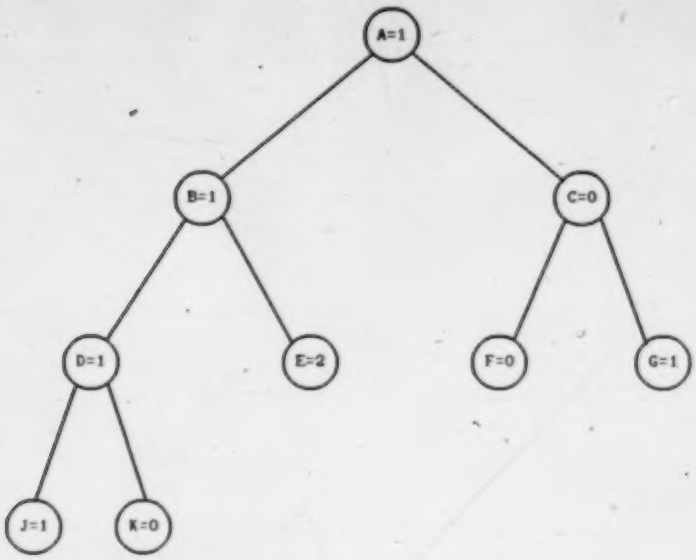

Figure 2.2 A search tree.

Conspiracy numbers provide a measure of the difficulty to change the current minimax value of a node. In Figure 2.2, assuming the root is a maximum node, how many leaf nodes in the tree have to change their value, as a result of being searched deeper, to cause the value at the root (*Vroot*) to become 2? The simplest way would be if node *J*'s value changed to 2. Another way would be for both nodes *F* and *G* to change their values appropriately. Nodes *F* and *G* form a *set of conspirators* for increasing *Vroot* to 2; both have to *conspire* to achieve this result. Node *J* also forms a set of conspirators for increasing *Vroot* to 2; in this case the *minimal* set. The minimum number of leaf nodes that must conspire to change *Vroot* to a specific value is called the *conspiracy number (CN)* for that value. Figure 2.3 shows the conspiracy numbers for node A, along with the minimal set of conspirators, for several possible node values.

Value	CN	Nodes to Change
-3	2	(E and (F or G))
-2	2	(E and (F or G))
-1	2	(E and (F or G))
0	1	(E or J)
1	0	
2	1	(J or K)
3	2	(E and (J or K)) or (F and G)

Figure 2.3 Conspirators.

It turns out that there are simple recursive relations for calculating the conspiracy numbers of a node from the conspiracy numbers of its descendants. In what follows, let m denote the minimax value of a node and v denote the value we would like to change m to.

At a leaf node, changing m to any other value requires a conspiracy of only that node itself, and hence has a conspiracy number of 1. If we do not want to change the node's value, then no conspiracy is required and the conspiracy number is 0. If the leaf node is also a terminal node,[5] then there is no way to change its value and a conspiracy number of ∞ is assigned. Hence, the conspiracy numbers for a leaf node are:

$$CN(v) = \begin{cases} 0 & \text{if} \quad v = m \\ 1 & \text{if} \quad v \neq m \\ \infty & \text{if} \quad \textit{terminal node} \end{cases}$$

At a maximizing interior node, to increase the value to v requires only one son to change its value to v. If the conspiracy number for each son has already been calculated, then the minimum number of conspirators required to increase the node to v, $\uparrow CN(v)$, is just the minimum number of conspirators to increase one of the sons to v. This yields the following relation:

$$\uparrow CN(v) = \begin{cases} 0 & \text{for} \quad \textit{all} \quad v \leq m \\ \underset{\textit{all sons } i}{MIN} \uparrow CN_i(v) & \text{for} \quad \textit{all} \quad v > m \end{cases}$$

[5] A terminal node represents the end of a game and therefore cannot have any successors.

To decrease the node's value to v, $\downarrow CN(v)$, requires all sons whose value is greater than v to decrease their value to v. Given the minimal set of conspirators for decreasing each son to v, *all* members of each of these sets must conspire together to decrease the node's value to v. Therefore:

$$\downarrow CN(v) = \begin{cases} 0 & \text{for } all \quad v \geq m \\ \sum_{all\ sons\ i} \downarrow CN_i(v) & \text{for } all \quad v < m \end{cases}$$

For a minimizing interior node, the following dual relations apply:

$$\uparrow CN(v) = \begin{cases} 0 & \text{for } all \quad v \leq m \\ \sum_{all\ sons\ i} \uparrow CN_i(v) & \text{for } all \quad v > m \end{cases}$$

$$\downarrow CN(v) = \begin{cases} 0 & \text{for } all \quad v \geq m \\ MIN_{all\ sons\ i} \downarrow CN_i(v) & \text{for } all \quad v < m \end{cases}$$

Figure 2.4 shows the conspiracy numbers for node values in the range [-3, 3] at each node. For convenience, conspiracy numbers for increasing and decreasing the node's value appear merged into one vector. This can be done because $\downarrow CN(v)$ is zero for those node values where $\uparrow CN(v)$ is non-zero and $\uparrow CN(v)$ is zero for those node values where $\downarrow CN(v)$ is non-zero. It is worth noting that if $v < w$ then $\uparrow CN(v) \leq \uparrow CN(w)$ and $\downarrow CN(w) \leq \downarrow CN(v)$. Also, given a set of conspirators for changing the value of a node to v, $(v \neq m)$, this same set can conspire to change the node to any value between m and v.

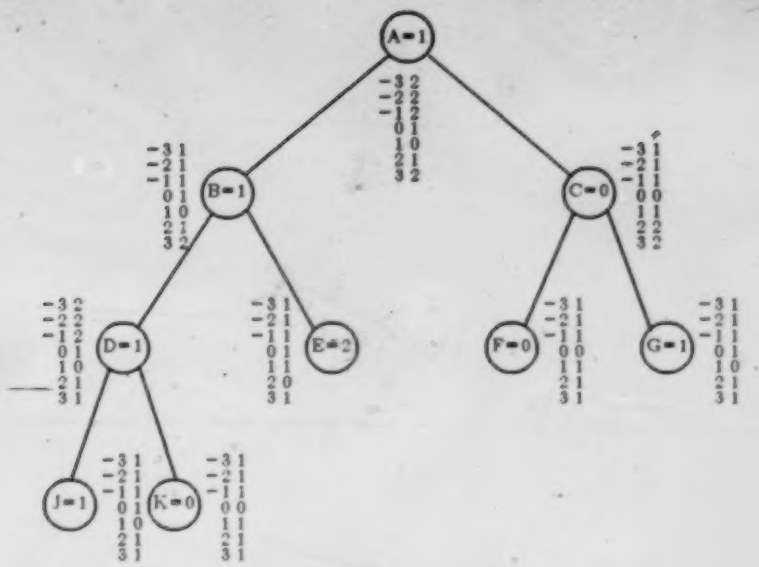

Figure 2.4 Conspiracy Numbers for the search tree.

2.3. Computing the Range of Likely Root Values

Conspiracy numbers at the root node, combined with a *conspiracy threshold (CT)*, are used to compute a range of likely root values. A conspiracy threshold specifies the minimum number of conspirators required before we consider it unlikely a node can take on that value. The range of likely root values is denoted by [*Vmin*, *Vmax*] where *Vmin* is the lower bound value, *Vmax* is the upper bound value, and *Vroot* is a member of this range. Values outside this range are considered unlikely because their corresponding conspiracy numbers are greater than or equal to CT, whereas values inside this range are considered likely root values because their conspiracy numbers are less than CT.

To compute a value for *Vmax*, the algorithm must find a node value v such that:

$$\uparrow CN(v) < CT \text{ and } \uparrow CN(v+1) \geq CT.$$

Similarly, to compute a value for *Vmin*, the algorithm must find a node value v such that:

$$\downarrow CN(v) < CT \text{ and } \downarrow CN(v-1) \geq CT.$$

It is important to notice that when $CT=1$, $Vmax = Vroot = Vmin$ (i.e. the range of likely root values consists of exactly one value which is *Vroot*). That is because the only value for which a conspiracy number is 0 is the current node value. It is therefore only practical to consider conspiracy thresholds of at least 2.

2.4. Terminating the Search

The algorithm continues to search until it has narrowed the range of likely root values to the point where $Vmax - Vmin \leq \Delta$. In practice, we usually continue searching until all but one root value has been ruled out (i.e. $\Delta = 0$); at this point, further search is unlikely to change the root's minimax value. When $\Delta = 0$ and the algorithm's condition for terminating the search evaluates to true, we often say that the algorithm has *converged* on a single root value.

Of course, this does not guarantee the root's minimax value to be correct, but the confidence in it can be measured by the conspiracy threshold: the higher the threshold, the greater the confidence in the root's minimax value. But one side effect of a higher threshold is that the amount of work required to narrow the range of likely values is larger because the initial range will be wider.

2.5. Strategies for Narrowing the Range of Likely Root Values.

Given a range of likely root values, how do we rule out all but one of them? The obvious way is to rule them out one by one, starting with either *Vmax* or *Vmin*. To rule out *Vmin* or *Vmax*, the algorithm tries to increase the corresponding conspiracy number to at least *CT*. This is done by "proving" that a member of the minimal conspiracy set will not conspire with the other members of the set to help change the value of the root node to either *Vmin* or *Vmax*.

During each step of the tree growth procedure, the algorithm must choose to

either *rule_out_Vmax* or *rule_out_Vmin*. Faced with these two alternatives, it chooses to attempt to rule out the value that is furthest from *Vroot*. If both are equidistant from the root value, it then arbitrarily chooses to *rule_out_Vmin*. Figure 2.5 summarizes the above decision.

```
if (Vmax - Vroot) > (Vroot - Vmin)
     rule_out_Vmax
else
     rule_out_Vmin
```

Figure 2.5 Choosing a strategy.

2.6. Selecting a Leaf Node for Expansion

Having made a decision to *rule_out_Vmax*, for example, a leaf node from the minimal set of conspirators must be found to search one ply deeper (or *expanded*). To find this node, the algorithm descends from the root using the following procedure:

a) at a maximizing node

Only one successor node must increase its value to *Vmax* for the root node to do likewise. The most likely branch is the one requiring the least number of conspirators to increase it to *Vmax*. After computing *CN*(*Vmax*) of each successor, choose the successor node requiring the minimum conspirators. If more than one branch has the minimum, arbitrarily choose the left-most one.

b) at a minimizing node

Here there may be many descendant nodes that have to increase their value to increase this node to *Vmax*. Each such branch contains conspirators which together form the set of conspirators to increase this node to *Vmax*. Again the algorithm can choose to traverse any of the appropriate branches and we arbitrarily choose to take the left-most one.

Having reached a leaf node, that node is expanded (i.e. searched one ply deeper). Since each descendant may yield a favorable or unfavorable assessment, the descendants are ordered according to the results of their evaluation. By putting the more favorable descendants first, this increases the chances that the left-most descendant is the best, justifying the above choices. The minimax value and conspiracy numbers are passed back up the tree, resulting in new numbers along the path from the root to the leaf node.

What is being accomplished by expanding this node? If we are successful at increasing the value of this node to *Vmax*, then the number of conspirators in this set has been decreased by one and therefore other members of the set can be expanded to see if they will conspire successfully. If the value is less than *Vmax* and the expanded node is minimizing, then we may have been successful at increasing the number of conspirators at the root (i.e. increased the minimal set of conspirators). The number of conspirators may have reached *CT*, resulting in a narrowing of the range of likely values at the root. At a maximizing expanded node with a value less than *Vmax*, nothing has been accomplished towards ruling out *Vmax*.

A dual strategy exists for ruling out *Vmin*. This tree growth procedure was McAllester's original proposal. Figure 2.6 illustrates the procedures for finding a leaf node when ruling out *Vmin* and when ruling out *Vmax*. The procedures, called *rule_out_Vmin* and *rule_out_Vmax*, find and return one leaf node for expansion.

```
rule_out_Vmax( Vmax )
{
    current_node = root_node
    add root_node to search path
    node_type = MAX
    while ( current_node is not a leaf node ) do {
        if ( node_type = MAX ) {
            compute CN(Vmax) for each son
            determine left-most son with minimum such value
            current_node = this son
            add current_node to search path
            node_type = MIN
        } else {
            determine left-most son where CN(Vmax) > 0
            current_node = this son
            add current_node to search path
            node_type = MAX
        }
    }
    return( current_node )
}

rule_out_Vmin( Vmin )
{
    current_node = root_node
    add root_node to search path
    node_type = MAX
    while ( current_node is not a leaf node ) do {
        if ( node_type = MAX ) {
            determine left-most son where CN(Vmin) > 0
            current_node = this son
            add current_node to search path
            node_type = MIN
        } else {
            compute CN(Vmin) for each son
            determine left-most son with minimum such value
            current_node = this son
            add current_node to search path
            node_type = MAX
        }
    }
    return( current_node )
}
```

Figure 2.6 Procedures for finding leaf nodes.

2.7. An Example

For this simple example, let the conspiracy threshold CT be 2 and let the range parameter Δ be 0. Also assume a maximum legal node value of 3 and a minimum legal node value of -3. For convenience though, all nodes in this example will be assigned a value of 0. Figure 2.7 illustrates the initial tree which consists of only the root node.

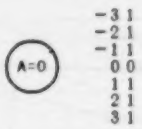

Figure 2.7 The initial tree.

The algorithm begins by computing *Vmin* and *Vmax*. Since there is only one node in the tree (i.e. a leaf node), the conspiracy number for each legal node value other than zero is 1. Hence *Vmin* is -3 and *Vmax* is 3. Since the algorithm has not converged on a single root value yet, it must continue expanding the tree. It chooses to rule out *Vmin*, even though both *Vmin* and *Vmax* are equidistant from the current root value. The routine rule_out_Vmin obviously chooses node *A* for expansion. Searching *A* 1-ply deeper yields 2 new states (*B*,*C*), each of which are given a static evaluation score of 0. The resulting tree after node *A* is expanded is depicted in figure 2.8.

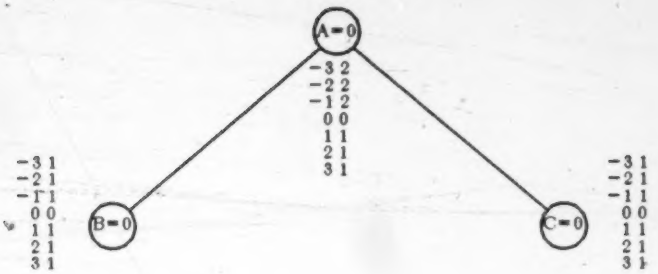

Figure 2.8 The tree after one expansion.

The algorithm now recomputes *Vmin* and *Vmax*, which become 0 and 3 respectively. This time it chooses to rule out *Vmax*. The routine rule_out_Vmax starts at node *A*, and since this node is already expanded, it must choose to descend down to one of its sons. Since the conspiracy number for *Vmax* is 1 at both sons, it arbitrarily chooses to descend down to node *B* (i.e. the left most son). Since node *B* is a leaf node, rule_out_Vmax returns it. Node *B* is then expanded, giving 2 new nodes *D* and *E*. The resulting search tree is depicted in figure 2.9.

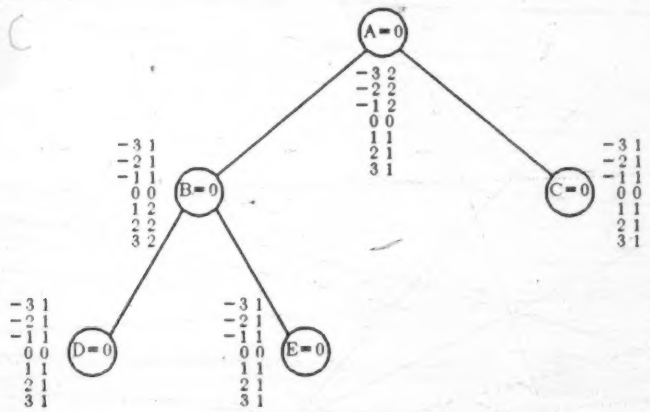

Figure 2.9 The tree after two expansions.

The algorithm again recomputes *Vmin* and *Vmax*, which remain 0 and 3 respec-

tively. Since *Vmax* is further from *Vroot* than *Vmin* is, the algorithm chooses to rule out Vmax. The routine rule_out_Vmax again starts at node *A* and must again decide which son to descend to. This time the conspiracy numbers for *Vmax* at nodes *B* and *C* are 2 and 1 respectively. Therefore rule_out_Vmax chooses to descend to node *C*, and since node *C* is a leaf node, rule_out_Vmax returns it. Node *C* is finally expanded, giving 2 new nodes *F* and *G*. The resulting search tree is depicted in figure 2.10.

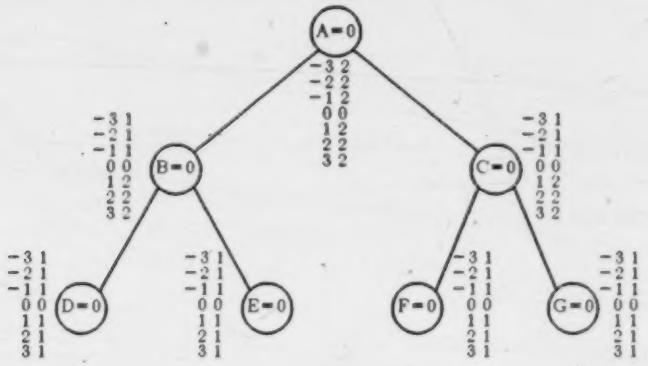

Figure 2.10 The tree after three expansions.

The algorithm again recomputes *Vmin* and *Vmax*, which now become both 0. This time, the termination condition is true, the algorithm has converged on a single root value of 0, and the search is stopped. Note that if a minimax algorithm with a depth limit of 2 would have been used here, the same tree would have resulted.

2.8. Comments

A few comments about the Conspiracy Numbers algorithm are in order now. First, conspiracy numbers are not probabilities nor should they be thought of as representing probabilities. This may have been confusing because of the terminology used to describe the algorithm. A conspiracy number is simply a *measure* of the difficulty to change the current minimax value of a node to another value. To compute the probability of this occurring, one would have to consider all the different

conspiracy sets for changing the current minimax value of the node to this other value along with the probability distributions at the leaf nodes.

Secondly, the algorithm seems to assume that all leaf nodes within a conspiracy set have an equal chance of changing their value to the desired value. In other words, when the algorithm computes conspiracy numbers, it does not take into account that some leaf nodes (i.e. conspirators) are highly unlikely to change their value from, for example, an extremely low value to an extremely high value. Conspiracy numbers can better reflect the difficulty to change the current minimax value of a node if conspiracy numbers at leaf nodes better reflect the difficulty of changing their values.

In deciding which strategy to pursue (i.e. rule_out_Vmax or rule_out_Vmin), there is no real good reason why the algorithm chooses to rule out the value furthest from V_{root}; it just seems to be a good heuristic. A better rule may be to rule out the value that has the highest conspiracy number (see figure 2.11). This way the algorithm always chooses the value that is easiest to rule out. However, when the algorithm is to search until it has ruled out all but one likely root value, it probably does not matter which rule for choosing a strategy we use.

```
if ↑CN(Vmax) > ↓CN(Vmin)
        rule_out_Vmax
else
        rule_out_Vmin
```

Figure 2.11 Alternative rule for choosing a strategy.

Finally, it should be apparent that this is a depth-first algorithm that does not use a depth limit. When the algorithm descends the tree in search of a leaf node to expand and has a choice as to which son to descend to, it always chooses the left most one. For the algorithm to function efficiently, it requires that the more favorable sons appear first. But even then there may be a considerable amount of wasted effort. For example, the algorithm may spend a considerable amount of time to find out that the

first two sons do not change their values to V_{max}, whereas the third son achieves this value much quicker. That is, more breadth in the algorithm may be beneficial in some circumstances.

Chapter 3

Implementing Conspiracy Numbers

At the heart of the Conspiracy Numbers algorithm are values we call conspiracy numbers. As we saw in Chapter 2, these numbers are used in two aspects of the algorithm: traversing the search tree to find a leaf node to expand, and computing values for *Vmax* and *Vmin*.

To do either of these tasks, we must compute conspiracy numbers for different node values at many nodes in the tree. But when the tree becomes large, computing them at a node near the top of the tree (eg. the root node) may become an expensive operation. This is true because the conspiracy number at a node is a function of the conspiracy numbers at each son which, in turn, are a function of conspiracy numbers of their sons, and so on. In other words, the entire tree must be traversed in order to compute one such value at the root.

Traversing the entire tree to compute a conspiracy number is necessary when none are stored at any nodes. It is however possible to store conspiracy numbers at each node and thus significantly reduce the time required to recompute them. One such scheme would require that we store a conspiracy numbers for each legal node value at each node. The problem then becomes one of maintaining them efficiently.

The main issue involved in implementing this algorithm is the space-time tradeoff. If no conspiracy numbers are stored at the nodes, the amount of memory required to store the search tree is minimized. Assuming that a major factor affecting the performance of the algorithm is the time it takes to compute these numbers, it is not yet obvious whether storing them at all nodes will have a positive impact on its performance because they must be continuously updated as the search tree grows.

To investigate this tradeoff, three implementation versions are presented in this chapter. The first version is called a Basic Implementation; it is probably the simplest

because it does not store any conspiracy numbers at the nodes. The second version is called a Storage Intensive Implementation; it saves conspiracy numbers for all possible node values at each node. The third version is called a Compromise Implementation; it stores conspiracy numbers for some important node values at all nodes. Experimental results involving these implementations will be included in the next chapter.

3.1. A Basic Implementation

The primary motivation for this implementation version is the concern for space. Since search trees for some games (such as chess) tend to get very large, minimizing the memory requirements can be important. However, another reason for implementing this version first is because it is the simplest and provides a basis for the other implementations to build and improve on.

A pseudo-code description of this version appears in Appendix 1. It is written using C-like programming constructs.

3.1.1. The Data Structures

There are two major data structures needed by this algorithm: one representing a node, and another representing a path in the search tree from the root to a leaf node. The minimal data requirements for a node are its minimax value and a pointer to its sons. The structure of our node is as follows:

```
struct Node {
    short int value;
    short int numSons;
    struct Node *sons;
}
```

In this structure, the variable *sons* is a pointer to the first element of an array of Nodes, and the size of this array is specified in the variable *numSons*. This allows us to create arrays of various sizes, and permits us to store a variable number of son nodes efficiently in memory. Figure 3.1 provides an illustration of a simple tree that uses

this data structure.

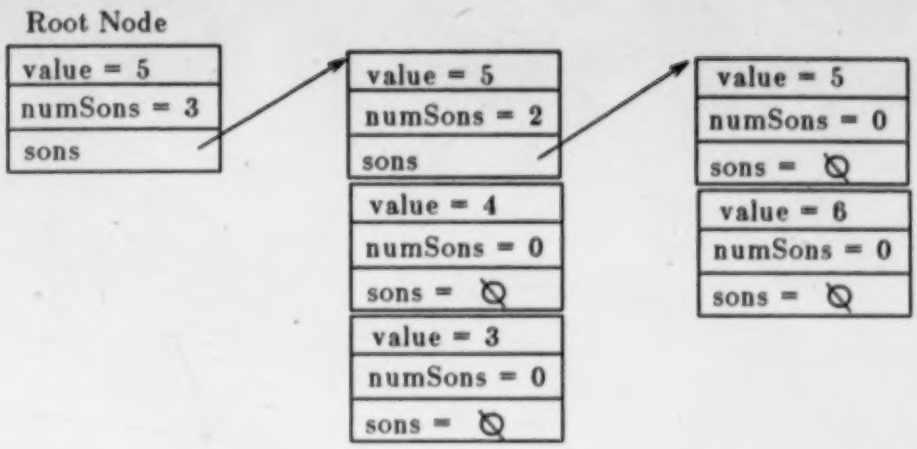

Figure 3.1 A search tree in the Basic Implementation.

To store a path between two nodes in the search tree, we will use an array of pointers to nodes:

```
struct Node *SearchPath [MAXDEPTH];
int PathDepth;
```

Typically, this structure is used for storing the path from the root to a leaf node, where the leaf node is the next node that the algorithm will expand. The global variable *PathDepth* is for storing the depth of this leaf node within the search tree, as well as its index in the array *SearchPath*. The main purpose in providing this structure is to allow the efficient updating of minimax values after each iteration.

3.1.2. Allowable Minimax Values

Even though there is no need to put a lower or upper bound on minimax values in this implementation version, we will nevertheless impose one. The main reasons for doing so are that the storage intensive version needs to provide space in each node for a conspiracy number for each possible node value, and for experimental purposes we would prefer that all three versions be as similar as possible. The upper bound will be denoted by the constant UB; the lower bound by LB.

Terminal nodes must also be flagged somehow. Since terminal nodes usually represent game states where a player has won, they will usually receive a score of UB or LB. To avoid introducing another field in the node structure just for flagging a terminal node, we will let these two values (i.e. UB and LB) function as flags. This also means reserving the use of these values to only terminal nodes.[6]

3.1.3. The Routines

The top level routine is shown in figure 3.2. It takes the conspiracy threshold (CT) as a parameter. The conspiracy threshold indicates the desired degree of confidence in the root node's value we wish to achieve before the search is terminated. This routine has only one minor difference with the top level algorithm presented in Chapter 2. The algorithm of Chapter 2 takes a second parameter, called the range parameter Δ, and terminates the search when $Vmax - Vmin \leq \Delta$. In this routine, we have simply fixed Δ so that it is zero. The result is that the algorithm searches until it has converged on a single root value. The resulting focus is then on finding a single *accurate* root value and not on finding a small enough range of plausible root values. This is especially necessary in game situations where alternative branches have to be narrowed down to as few choices as possible, and by finding only one root value certainly helps attain that goal. Note that narrowing down to one value does not mean that the algorithm has narrowed the choices down to one branch.

---

[6] Some terminal nodes also correspond to game states representing draws. Such nodes will typically receive a value of (UB - LB)/2 and must therefore be flagged differently. For simplicity, this case will be ignored here; it will however be dealt with in a later chapter.

```
Top_Level (CT)
int CT;
{
    int Vroot, Vmin, Vmax;
    struct Node *aNode;

    if (RootNode == NULL)
        Initialize-root();

    Vmin = Compute_Vmin (Vroot,CT);
    Vmax = Compute_Vmax (Vroot,CT);
    Vroot = RootNode -> value;

    while (Vmin ! = Vmax) do {
        /* find a path to a node to expand   */
        if (Vmax - Vroot) > (Vroot - Vmin)
            aNode = Rule_out_Vmax (Vmax);
        else
            aNode = Rule_out_Vmin (Vmin);

        /* descend to the node and expand it */
        Expand (aNode, PathDepth);

        /* recompute the minimax values,     */
        /* along with Vmin, and Vmax         */
        Update_minimax_values();
        Vmin = Compute_Vmin (Vmin,CT);
        Vmax = Compute_Vmax (Vmax,CT);
        Vroot = RootNode -> value;
    }
    return (Vroot);
}
```

Figure 3.2 The Top_Level Routine.

Since this implementation does not store conspiracy numbers at the nodes, we must recompute them every time they are needed. The function CN, which appears in Appendix 1, does this. This routine incorporates all the expressions for computing conspiracy numbers which were described in Chapter 2. It is a recursive function that traverses the entire sub-tree below a node in order to compute a conspiracy number for that node. It uses the value of two parameters, ***direction*** and ***node_type***, to identify which of the four expressions to apply: ***direction*** can take on either INCREASE or

DECREASE, indicating whether we wish to increase or decrease the node's value, and *node_type* can take on either MAX or MIN, indicating whether the node is a maximizing or minimizing node.

The routines for finding a leaf node to expand (i.e. Rule_out_Vmax () and Rule_out_Vmin()) are virtually identical to their algorithms which were presented in Chapter 2. They do, however, perform one additional task: they save the path from the root node to the leaf node in the global array SearchPath.

Recall that in order to compute a value for *Vmax*, we must find a node value v such that:

$$\uparrow CN(v) < CT \text{ and } \uparrow CN(v+1) \geq CT.$$

Finding such a value v is actually a simple procedure. Suppose we start with any value v. Then,

- if $\uparrow CN(v) < CT$, compute $\uparrow CN(v')$ for each $v' = v+1, v+2, \ldots$ until we find one that is $\geq CT$. When such a v' is found, we then know that $v'-1$ is the new value for *Vmax*.
- if $\uparrow CN(v) \geq CT$, compute $\uparrow CN(v')$ for each $v' = v-1, v-2, \ldots$ until we find one that is $< CT$. When such a v' is found, it becomes the new value for *Vmax*.

This is how the routine Compute_Vmax() finds a new *Vmax*. Computing a new *Vmin* can be done similarly.

But what is a good starting value for v? The objective in choosing a starting value is to minimize the time required to find a new *Vmin* or *Vmax*. To accomplish this, we must find a v which is as close to the new *Vmin* or *Vmax* as possible, because only then can we minimize the number of v' tested. From observations, we have noticed that both *Vmin* and *Vmax* change gradually. In fact, we have also noticed that *Vmin* and *Vmax* often do not change from one iteration to another. So a good starting value would be the old *Vmin* or *Vmax*.

The function Expand() illustrated in Appendix 1 is just a simple expansion routine that generates a number of sons and assigns them a value. The important thing here is that these sons have to be sorted so that the "best sons" appear at the beginning. That is:

- if a MAX node was expanded, then the sons should be sorted in descending order according to their value.
- if a MIN node was expanded, then the sons should be sorted in ascending order according to their value .

The prime motivation for this is that when the algorithm traverses the search tree looking for a new leaf node to expand, it always traverses the left most branch when alternative branches could have been taken. Putting the "best sons" near the beginning enhances their chances of being searched deeper, increasing the algorithm's chances of finding a winning move.

Updating minimax values can be done in an efficient manner. When a node changes its minimax value, the only nodes that are affected are its ancestor nodes. When a leaf node is expanded, its minimax value may change, which means the minimax value for its ancestors may need updating. A simple approach for updating these minimax values, which is employed by the routine Update_minimax_values(), is to compute the minimax value of each node along the search path starting at the expanded node, working our way up towards the root node.

3.1.4. Calling the Top Level Routine

Obviously the routine that calls Top_Level will be some sort of driver routine. When calling Top_Level, the driver must supply it with a conspiracy threshold (CT). Two factors will influence the value which should be chosen for CT. These factors are the time required for the algorithm to converge onto a node value for a certain CT, and the confidence in this node value given the CT that was used to find it. Generally, the higher the conspiracy threshold that is used, the longer it takes for the algorithm to converge on a single node value. Also, the higher the conspiracy threshold, the more confidence we can have in the final root value.

It is difficult to balance these two factors because it is impossible to predict the time it will take the algorithm to converge for a certain CT and because it is impossible to quantify the degree of confidence based on a CT. However, there is an approach that allows the degree of confidence in the root value to increase over time. It starts out by assigning CT a low value such as 2. Once the algorithm converges for this conspiracy threshold, it is bumped up to 3, and continues so forth until a specified time limit is reached. This makes it easier for the algorithm to converge in the specified time limit, as well as allow us to comparatively quantify degrees of confidence with respect to CT. This method is analogous to iterative deepening which is used for the alpha-beta and A* search algorithms [Kor85, SlA83].

3.2. A Storage Intensive Implementation

The primary motivation for this implementation is to see if we can increase the performance of the algorithm if conspiracy numbers are stored at each of the nodes. This implementation differs from the basic implementation in that it stores conspiracy numbers for each node value at all nodes. These conspiracy numbers have to be continuously updated as the minimax tree grows and changes, and hence increases the complexity and difficulty of implementing this version.

A pseudo-code description of this version appears in Appendix 2.

3.2.1. The Data Structures

The structure of a node now must include fields for storing conspiracy numbers. But what is the best way for storing all these conspiracy numbers? One way would be to have two vectors, call them *increaseCN* and *decreaseCN*, which store the conspiracy numbers for all possible node values. The vector *increaseCN* would then contain conspiracy numbers for increasing a node's value to each of the possible node values, whereas the vector *decreaseCN* would contain the conspiracy numbers for decreasing the node's value. However, if the node's value is V, then:

- $\uparrow CN(v) = 0$ if $v \leq V$
- $\downarrow CN(v) = 0$ if $v \geq V$

so that more than half the total values in *increaseCN* and *decreaseCN* combined are 0. Specifically, all conspiracy numbers in *increaseCN* which are to the left of the conspiracy number corresponding to V will be 0, and similarly all conspiracy numbers in *decreaseCN* which are to the right of the conspiracy number corresponding to V will be 0. It is therefore more efficient to store conspiracy numbers in a single vector; we will call this new vector *CN*. The structure of Node is enhanced as follows:

```
struct Node {
    short int value;
    short int numSons;
    struct Node *sons
    unsigned char CN[UB-LB+1];
}
```

Getting the correct conspiracy number from this node involves checking two things: the direction we wish to change the node's value in, and the relationship between the node's value and the value we wish to change it to. Specifically, let V = the node's value; then

- if we wish to increase the node's value to v, where $v \leq V$, then the conspiracy number is 0
- if we wish to decrease the node's value to v, where $v \geq V$, then the conspiracy number is 0,
- otherwise use v to index the vector CN stored at the node.

Note from the structure above that we are only allotting one byte for each conspiracy number. In previous discussions, it was mentioned that the algorithm converges slower for larger conspiracy thresholds (CT) than it does for smaller ones. In our experiments, rarely did we have the algorithm converge for conspiracy thresholds larger than 20. Since we are working with such small conspiracy thresholds, there is no need to store large conspiracy numbers and a byte, which provides a range of 256 possible values, is more than a sufficient amount to store conspiracy numbers in.

We can make additional savings on storage by reducing the amount of information stored in a leaf node. Recall that the conspiracy numbers at leaf nodes can only take on the value 0 or 1. The vector CN at leaf nodes will contain 1's for all entries except for the conspiracy number corresponding to the leaf node's value (which is 0). Since computing conspiracy numbers at leaf nodes is simple and quick, we can eliminate the vector CN from leaf nodes. To do this, the node structure will be broken into two parts as follows:

```
struct Node {
      short int value;
      struct Extension *part 2;
}

struct Extension{
      short int numSons;
      struct Node *sons;
      unsigned char CN[UB-LB+1];
}
```

Leaf nodes will only consist of the first structure (i.e. Extension = NULL), whereas interior nodes will obtain the second part as an extension (at the time it is expanded).

Since most nodes in a minimax tree are leaf nodes, significant storage savings can be obtained using the above structure. Figure 3.3 provides an illustration of a simple tree that is identical to the tree of figure 3.1, but uses the data structure for this implementation.

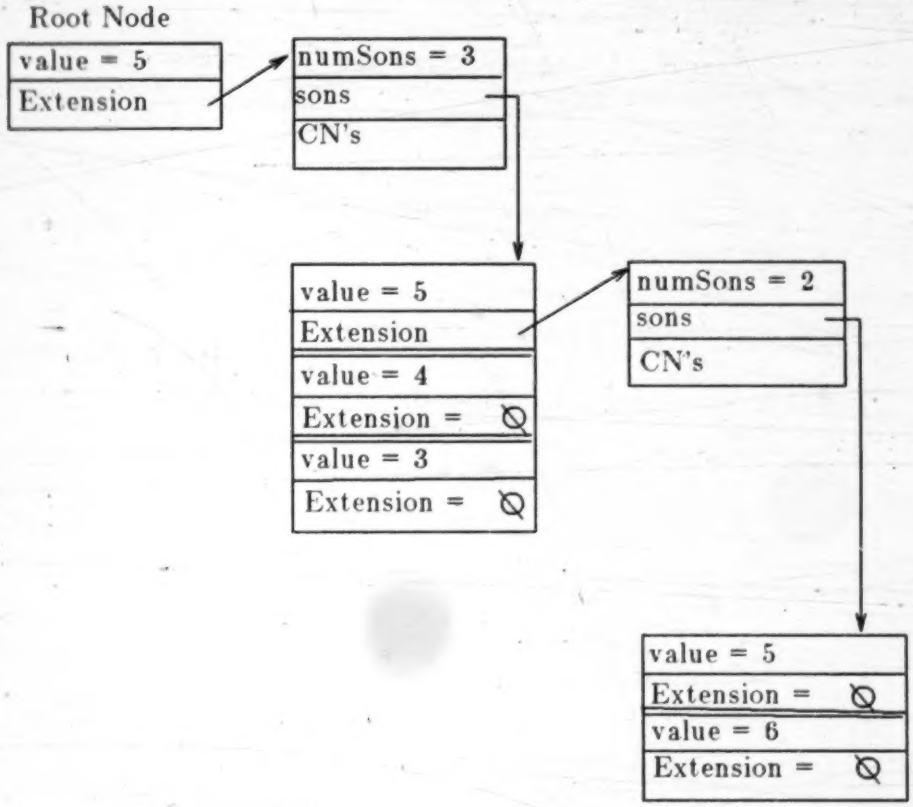

Figure 3.3 A search tree in the Storage Intensive Implementation.

Getting the correct conspiracy number from a node now involves one additional task: checking if the node is a leaf node. If the node is a leaf node, then the expressions presented in Chapter 2 for computing conspiracy numbers at leaf nodes must be applied. If the node is not a leaf node, then the previous procedure for getting the correct conspiracy number must be applied.

3.2.2. Maintaining the Conspiracy Numbers

All routines used in the Basic Implementation version can be carried over with some modifications needed because of the new node structure. However, an additional routine that updates the conspiracy numbers is required. This routine will be called Update_CN(); it is called from the Top_Level routine, immediately after the minimax values have been updated.

Whenever a node's conspiracy numbers change, the only other conspiracy numbers that this may affect are those of its ancestor nodes, as was the case with minimax values. Since the expanded node will certainly have its conspiracy numbers change, its ancestor nodes may require their conspiracy numbers be updated. As was the case with updating minimax values, updating conspiracy numbers starts at the expanded node and works its way up toward the root node. At each node, conspiracy numbers are recomputed for all possible node values, before proceeding to the next node.

Traversing the search tree to find a leaf node to expand can be made more efficient now that conspiracy numbers are stored at each node. For example, consider the routine Rule_out_Vmax() which relies on the conspiracy numbers of *Vmax* for traversing the tree. At maximizing nodes, the conspiracy number for *Vmax*, which is stored at the node, is the minimum of the conspiracy numbers for *Vmax* at each son. Comparing the conspiracy number for *Vmax* with each of the sons' conspiracy number for *Vmax*, the left most branch to be traversed next can be identified quicker. A similar case exists for the routine Rule_out_Vmin().

3.3. A Compromise Implementation

This version, which we call a compromise implementation, will combine several features from the previous two implementation versions. It will store conspiracy numbers for only some node values at each node, which means that conspiracy numbers for other node values must be recomputed whenever they need to be used. Obviously we want to store the most often used conspiracy numbers at each node, and thereby look for a balance in the tradeoff between space and time. To implement this version, we must look at two issues: for what node values should we store conspiracy numbers, and how do we maintain these conspiracy numbers.

A pseudo-code description of this version is appears in Appendix 3.

3.3.1. Which Conspiracy Numbers to Store

Before proceeding, note that the routine that computes conspiracy numbers in the Basic Implementation traverses the entire search tree. When the conspiracy number for *Vmax* is computed at the root node, this routine visits every node and computes its conspiracy number for *Vmax*. Conspiracy numbers, for a value of *Vmax*, can be stored at each node and then updated when necessary. If *Vmax* rarely changes, then occasionally updating these numbers may be more efficient than recomputing them every time.

The goal in choosing a set of values to store conspiracy numbers for is to reduce the amount of recomputation of them during each iteration. In deciding what node values to store conspiracy numbers for, it is best to look at how they are used in the algorithm. It turns out that conspiracy numbers are used in performing only two types of tasks: traversing the search tree, and computing values for *Vmax* and *Vmin*.

When the routine Rule_out_Vmax() traverses the search tree, it relies on the conspiracy number for *Vmax*. If the conspiracy number for *Vmax* were stored at each node, it would eliminate the necessity of recomputing this value at many nodes as

Rule_out_Vmax() traverses the tree. Similarly, storing the conspiracy number for *Vmin* at each node would make Rule_out_Vmin() more efficient. Therefore, two values for which conspiracy numbers should be stored for at each node are *Vmax* and *Vmin*.

Recall that when computing a value for *Vmax*, the routine Compute_Vmax() finds a value v such that $\uparrow CN(v) < CT$ and $\uparrow CN(v+1) \geq CT$. The first value of v that is tested by this routine is always the current value of *Vmax*, and once such a v is found it becomes the new *Vmax*. Therefore, the minimum number of conspiracy numbers that are tested by Compute_Vmax() is two. This case occurs when *Vmax* does not change, and the two conspiracy numbers are those for the current values of *Vmax* and *Vmax*+1. If *Vmax* does not change too often, storing conspiracy numbers for *Vmax* and *Vmax*+1 at nodes will make Compute_Vmax() more efficient. Similarly, Compute_Vmin() can be made more efficient if conspiracy numbers for *Vmin* and *Vmin*-1 are stored at the nodes. Therefore, in addition to storing conspiracy numbers for *Vmax* and *Vmin* at the nodes, conspiracy numbers for *Vmax*+1 and *Vmin*-1 should also be stored at the nodes.

Let *Vmax*1 = *Vmax*+1 and *Vmin*1 = *Vmin*-1, and let the conspiracy numbers for the values of *Vmin*1, *Vmin*, *Vmax*, *Vmax*1 be denoted by *CNvmin*1, *CNvmin*, *CNvmax*, *CNvmax*1, then the structure of a node for this implementation version is:

```
struct Node {
        short int value;
        short int numSons;
        struct Node *sons;
        unsigned char CNvmin, CNvmin1;
        unsigned char CNvmax, CNvmax1;
}
```

A diagram illustrating a search tree using this data structure would be similar to figure 3.1 and is therefore omitted here.

3.3.2. Maintaining the Conspiracy Numbers

When computing conspiracy numbers, it rarely happens that the entire search tree has to be traversed. For example, when computing the conspiracy number for increasing a node's value to *Vmax*, if the node's value is already greater than or equal to *Vmax* the answer is already known to be 0, and hence the nodes below it need not be examined (i.e. traversed any further). In what follows, call those nodes which need to be examined by a routine that computes a conspiracy number "relevant" nodes, and the others "irrelevant" nodes. Note that the same node may be both irrelevant and relevant with respect to different node values (such as *Vmin* and *Vmax*).

There are two choices available in maintaining the conspiracy numbers at the nodes. The first choice is making sure that all conspiracy numbers at all nodes are correct. The second choice is allowing conspiracy numbers for some values to be incorrect at those modes that are irrelevant with respect to these values.

Keeping all conspiracy numbers at all nodes correct is the least complex to implement. However, it may also be the least efficient one. Suppose that *Vmax* changes. This means that the conspiracy numbers for *Vmax* and *Vmax*1 must be changed for all nodes in the tree. If the tree becomes large, we may want to avoid making changes to conspiracy numbers at irrelevant nodes and therefore allow incorrect conspiracy numbers to exist at these nodes. Although more complex, we believe that the second alternative is more efficient and therefore worthwhile to implement.

Both this version and the Storage Intensive version store conspiracy numbers at the nodes, and therefore appear to be similar in their implementation. However, since this version only stores the conspiracy numbers for a subset of node values, they do have some differences. This version differs primarily by how it has to update the conspiracy numbers of the nodes.

3.3.3. Updating the Conspiracy Numbers when Vmin or Vmax Change

Recall that the values for *Vmin* and *Vmax* can only be altered through the routines Compute_Vmin() and Compute_Vmax(). Note that one way in which the conspiracy numbers that are stored at the nodes can change is if either *Vmin* or *Vmax* change. Therefore, these two routines must also be responsible for initiating the computation of new conspiracy numbers that will be stored at the nodes when *Vmin* or *Vmax* change. Specifically, Compute_Vmin() will be responsible for initiating the updating of *CNvmin* and *CNvmin*1 at all nodes, while Compute_Vmax() will be responsible for initiating the updating of *CNvmax* and *CNvmax*1. To examine how updates are initiated and how updates can be done efficiently, we will examine what happens when *Vmax* changes inside the routine Compute_Vmax(). Updates to *CNvmin* and *CNvmin*1 are handled similarly in Compute_Vmin() when *Vmin* changes.

Initially, assume that *CNvmax* is correct at all nodes that are relevant with respect to the value of *Vmax*, and similarly *CNvmax*1 is correct at all nodes that are relevant with respect to the value of *Vmax*1. Also, assume that *CNvmax*1 is less than CT at the root node, and therefore *Vmax* must increase. What are the sequence of steps that must be taken in order to find a new *Vmax*?

First, *Vmax* is increased by one to the old value of *Vmax*1, and *Vmax*1 is implicitly increased by one. This should prompt a recomputation of *CNvmax* and *CNvmax*1 for all relevant nodes below the root node. Since the new *Vmax* is equal to the old *Vmax*1, the new value for *CNvmax* at any node will be equal to the old value of *CNvmax*1 at that node. Therefore, while computing new values for *CNvmax*1, the old value of *CNvmax*1 can be copied to *CNvmax* at each node that is visited and avoid a separate computation for *CNvmax*. This is what the routine CN_Vmax1() does. This whole process repeats itself until a *Vmax*1 whose conspiracy number is greater than or equal to CT is found.

Will *CNvmax* and *CNvmax*1 be correct at all relevant nodes after this process is completed? Note first that the set of relevant nodes with respect to *Vmax* is a subset of the set of relevant nodes with respect to *Vmax*1. In other words, when the routine that computes the new *CNvmax*1 traverses the tree, it will also visit all relevant nodes with respect to the new value of *Vmax* (which used to be the old *Vmax*1) and copy the each old *CNvmax*1 (which were assumed to be correct at these nodes) to *CNvmax*. Since all relevant nodes are going to be visited, and the appropriate action will be performed at these nodes, we know that *CNvmax* and *CNvmax*1 will be correct at all relevant nodes after this process is completed.

What happens when *CNvmax* is greater than or equal to CT? Compute_Vmax must decrement *Vmax* in this situation until a *CNvmax* that is less than CT is found. The process is similar to the above process but, instead of incrementing *Vmax*, we decrement it and instead of computing new values for *CNvmax*1 (and copying the old values to *CNvmax*) we compute new values for *CNvmax* (and copy the old values to *CNvmax*1).

But there will be one problem here. When computing *CNvmax* for a new value of *Vmax*, only nodes that are relevant to the value of *Vmax* will be visited, and these nodes form only a subset of the set of relevant nodes with respect to the value of *Vmax*1. Obviously we have to visit the remaining nodes that are relevant to the value of *Vmax*1, but not to the value of *Vmax*, and copy *CNvmax* to *CNvmax*1. Note that no new *CNvmax* must be computed at these remaining nodes.

Finding these nodes is simple. When computing *CNvmax*, a node whose value is equal to *Vmax* is normally not traversed any further, whereas when computing *CNvmax*1 it would be traversed. It is under such nodes that the rest of the relevant nodes with respect to the value of *Vmax*1 can be found. When nodes whose minimax value is equal to *Vmax* are encountered, simply traversing the tree to all relevant

nodes with respect to only *Vmax*1 and copying the old *CNvmax* (which is assumed to be correct) to *CNvmax*1 will solve this problem. Again, it can be shown the *CNvmax* and *CNvmax*1 are correct at all relevant nodes.

3.3.4. Updating the Conspiracy Numbers after an Expansion

The only other time updates to conspiracy numbers can occur is after the expansion of a leaf node. Only those nodes along the path from the root node to the expanded node, as well as the newly created sons, have to get their conspiracy numbers updated. This is done by the routine Update_CN(). This routine first computes the conspiracy numbers at the newly created sons, and then updates the conspiracy numbers for those nodes along the path, starting from the expanded node and working its way up toward the root node. At each node, conspiracy numbers for *Vmin*, *Vmin*1, *Vmax*, and *Vmax*1 are computed before proceeding to the next node.

Since conspiracy numbers at irrelevant nodes are not kept correct, there is one major problem that can occur when updating the conspiracy numbers in the manner described above. To illustrate this problem, assume that the algorithm is at a point where it has decided to try to rule out *Vmax* and has chosen a leaf node to expand. We know that all node values along the path from the root node to the leaf node must be less than *Vmax* before this leaf node is actually expanded. In fact, some of these nodes may have a minimax value that is less than or equal to *Vmin*. Let's assume that there is one node along this path whose minimax value is less than or equal to *Vmin*, and let's call it node A. [If more than one such node exists, choose as node A the one that is closest to the root node]. Recall that all nodes that are descendants of this node A are irrelevant with respect to the value of *Vmin*. Now suppose the leaf node is expanded, and this results in node A's minimax value increasing to above *Vmin*. This will result in descendants of node A becoming relevant with respect to the value of *Vmin*. However, these "new relevant" nodes will have incorrect values for *CNvmin*, and when the

above update procedure is applied these incorrect values may propagate up and corrupt the conspiracy numbers at previously relevant nodes. The only way to fix this is to find these "new relevant" nodes and correct their conspiracy numbers before the update procedure is started.

The occurrences of "new relevant" nodes can only result from the expansion of a leaf node. When expanding a leaf node while trying to rule out *Vmax*, we must be concerned about "new relevant" nodes with respect to *Vmin* and *Vmin*1 occurring. Similarly, when expanding a leaf node while trying to rule out *Vmin*, we must be concerned about "new relevant" nodes with respect to *Vmax* and *Vmax*1 occurring. Since the procedures for handling the cases dealing with "new relevant" nodes with respect to either *Vmin*, *Vmin*1, *Vmax*, or *Vmax*1 are identical, only one of them will be discussed.

The routine fixup_CNvmin() handles the case where "new relevant" nodes with respect to *Vmin* occur as a result of a leaf node being expanded while the algorithm tries to rule out *Vmax*. What this routine does is find where these "new relevant" nodes are and call the appropriate routines to correct their values for *CNvmin*.

Finding these "new relevant" nodes is simple. The routine fixup_CNvmin() traverses the path from the root node to the expanded node. When it comes across a node whose old *CNvmin* is 0, it knows that the algorithm for computing new values for *CNvmin* did not traverse any of this node's descendants and therefore their values for *CNvmin* are incorrect. It is at such a node that "new relevant" nodes may occur under. If the value of this node increases to a value that is greater than *Vmin*, "new relevant" nodes will appear under it and a routine for recomputing their conspiracy number for *Vmin* must be called.

More specifically, suppose that fixup_CNvmin() traverses to a node whose value for *CNvmin* was 0. Depending on what the node's new minimax value is, two possible cases arise:

- if the node's new value is less than or equal to *Vmin*, all descendant nodes remain irrelevant with respect to *Vmin* and therefore no further action needs to be taken, and
- if the node's new value is greater than *Vmin*, fixup_CNvmin() must call a routine which will recompute the conspiracy numbers for *Vmin* below this node.

As a result of this, conspiracy numbers for *Vmin* at all "new relevant" nodes below it will be correct, and conspiracy numbers for *Vmin* at irrelevant nodes will be incorrect (which does not matter anyhow). At this point, further traversal is not necessary and hence fixup_CNvmin() returns.

When the conspiracy numbers at "new relevant" nodes have been corrected, the procedure for updating the conspiracy numbers along the path from the root node to the expanded node can be safely started. The only nodes with incorrect conspiracy numbers are nodes that are irrelevant with respect to a value, but these conspiracy numbers can be simply prevented from propagating up and corrupting the conspiracy numbers at relevant nodes. To see why this is true, assume that there is a node along this path that is relevant with respect to *Vmin*, but whose descendants are irrelevant with respect to *Vmin*. Since this node's minimax value must be less than or equal to *Vmin*, the conspiracy number for *Vmin* at this node is automatically known to be 0, and does not depend on the conspiracy numbers of its irrelevant descendants. Therefore the conspiracy numbers at relevant nodes will be correct when the update routine finishes.

3.4. Conclusion

Three implementation versions which demonstrate the trade-off between speed and space were presented here. From our experience, we learned that computing conspiracy numbers represents the most computationally intensive part of the algorithm, and that conspiracy numbers were often needlessly recomputed. In order to reduce the recomputation of conspiracy numbers, it was decided that storing conspiracy numbers at each node may be of benefit.

The Basic Implementation and the Storage Intensive Implementation represented two extremes from opposites ends of the spectrum. The Basic Implementation stores no conspiracy numbers at the nodes whereas the Storage Intensive Implementation stores conspiracy numbers for each possible node value at the nodes. However, the Basic Implementation spends a lot of time recomputing conspiracy numbers needlessly, and the Storage Intensive Implementation spends a lot of time updating conspiracy numbers for several needless node values along the path of expansion. When the range of node values becomes large, the Storage Intensive Implementation may actually perform worse because updating all the conspiracy numbers along the path of expansion will consume a large amount of time. Also, limited memory will certainly affect the performance, as well as applicability, of the the Storage Intensive Implementation more than the Basic Implementation.

The Compromise Implementation tries to find a middle ground between the two previous implementations. Instead of storing conspiracy numbers for all node values, it only stores conspiracy numbers for the most important node values (i.e. *Vmin*, *Vmin*1, *Vmax*, *Vmax*1). This reduces the amount of updating of conspiracy numbers that is performed along the path of expansion, reduces the amount of needless recomputation of conspiracy numbers, and reduces the memory requirements for storing large minimax trees. This implementation version depends on the assumption that *Vmin*

and *Vmax* remain fairly stable for improved performance over the Basic Implementation.

But which implementation version is preferable? If the assumption about *Vmin* and *Vmax* remaining fairly stable is true, then the Compromise Implementation would be superior to the Basic Implementation. If the range of node values is large, the Compromise Implementation is preferable to the Storage Intensive Implementation because it requires less memory and the updating cost of conspiracy numbers along the path of expansion is less. If the range of node values is small, then the Storage Intensive Implementation may be preferable because it will probably perform faster than the others. As can be seen, several factors will influence our choice of a particular implementation. It is our experience, however, that there are only two practical choices: the Storage Intensive Implementation and the Compromise Implementation.

Chapter 4

Results for Random Trees

Initial experiments with the conspiracy numbers algorithm were done on randomly generated search trees. In our random trees, the number of sons that are created when a node is expanded and the node values of the leaf nodes will both be a function of random values. The reasons for using random trees is that they are easy to implement and are flexible, which allows us to do a variety of experiments. The objectives of these experiments were threefold: to compare the performances of the three implementation versions of the algorithm, to understand how some factors affect the performance of these versions, and to learn something about the algorithm's behavior. This chapter summarizes the results of our experiments.

4.1. Performance Results

How do the performances for the three implementation versions of the conspiracy numbers algorithm compare? Figure 4.1 illustrates the performances of these three versions.[7] In this graph, the number of nodes generated is plotted against time. The trees that they generated each had a constant branching factor of 5 and node values were integers in the range [-20,20]. Based on these results, we can conclude that the Basic Implementation version of the algorithm is inferior to the other two versions. The performances of the Storage Intensive Implementation and the Compromise Implementation are fairly close and require further experiments to differentiate between them more precisely.

[7] All experiments reported in this chapter where done on a SUN 3/75.

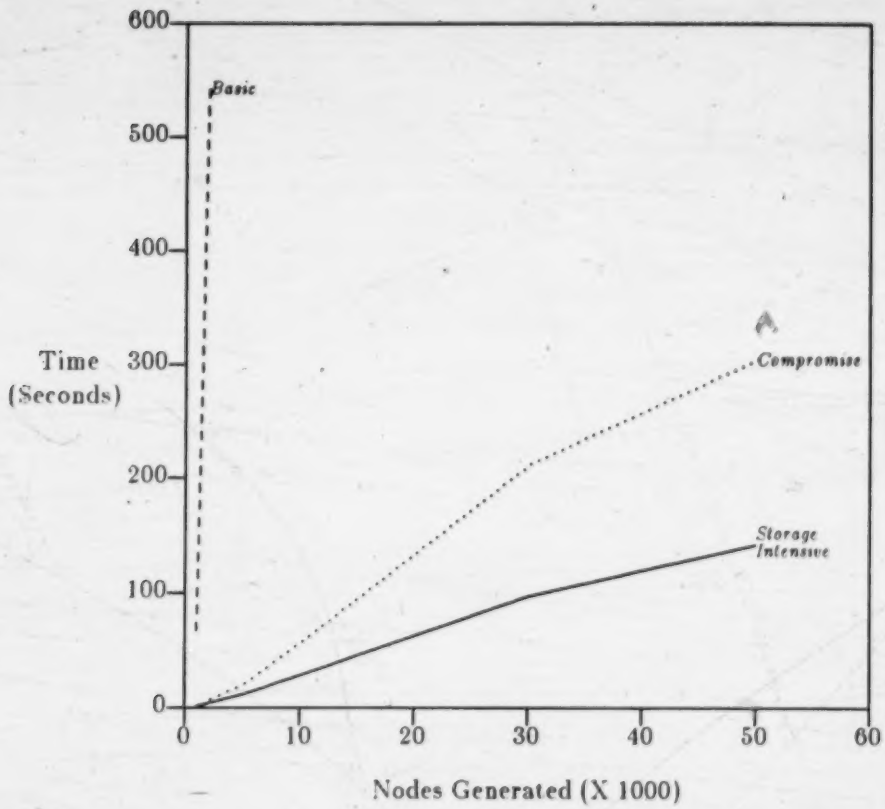

Figure 4.1 Initial Performance Comparisons.

Figure 4.2 shows for each implementation the breakdown of the total time that was spent searching one particular minimax tree. The breakdown of the total time is according to the type of tasks performed in each version. What this confirms is that the computation of conspiracy numbers accounts for the largest proportion of the total time. By concentrating improvements in this one area of the implementation (eg. saving conspiracy numbers at the nodes) we can significantly increase its performance as shown by the results in figure 4.1.

Basic Implementation		
Task	Time (sec.)	Percentage
Compute CNs	7427.63	99.2
Compute Vmin, Vmax	56.08	0.7
Rule_out_Vmax	5.55	0.1
Rule_out_Vmin	0.44	0.0
others	1.31	0.0
Totals	7491.01	100.0

Storage Intensive Implementation		
Task	Time (sec.)	Percentage
Update CNs	8.82	41.9
Accessing CNs	6.24	29.6
Rule_out_Vmax	3.58	17.0
Rule_out_Vmin	0.38	1.8
others	2.05	9.7
Totals	21.07	100.0

Compromise Implementation		
Task	Time (sec.)	Percentage
Update CNs	27.20	76.7
Accessing CNs	3.17	8.9
Rule_out_Vmax	3.16	8.9
Rule_out_Vmin	0.16	0.5
Compute new CNs	0.12	0.3
others	1.67	4.7
Totals	35.48	100.0

Figure 4.2 Profiles of the Implementations.

There is one minor flaw in these performance comparisons: they are not totally fair because each version was probably not optimally implemented. Improvements may be possible in each implementation that can increase these performances. Overall, the implementations that were used to derive the results in this chapter were roughly at the same *level of optimization*, and therefore the performance comparisons should be reasonably fair.

Since the Storage Intensive version and the Compromise version are the only two practical alternatives, let us focus our attention on only them. Factors that will affect

the performance of that part of the implementation that maintains the conspiracy numbers will obviously affect the overall performance of the implementation a great deal. Since the way each of these two versions maintains its conspiracy numbers at each node differs, these factors will also have different affects on each version and hence help differentiate them. The primary factors that affected these versions' performances were: the size of the range of legal node values, the average branching factor in a minimax tree, and the *average depth* of nodes in a particular tree.[8]

Figure 4.3 compares the performances for the Storage Intensive version and the Compromise version for several different range sizes. To generate these particular results, each version was run on several hundred different search trees. In this graph, the number of nodes expanded per second is plotted against the range size. These results clearly show that varying the range size affects the performance of the Storage Intensive version much more than the performance of the Compromise version. The Storage Intensive version has to maintain conspiracy numbers for the entire range of node values whereas the Compromise version has to only maintain conspiracy numbers for four node values (i.e. a constant number of conspiracy numbers regardless of the range size). In addition, these results also show that there is a range of size for which the Storage Intensive version has superior performance and a complementary range of size for which the Compromise version has superior performance. The boundary between these two ranges will of course vary because other factors (i.e. the branching factor and the average depth of nodes) will affect the performance of each version in different ways.

---

[8] We define the average depth of nodes in a particular tree as: $(\sum_{\text{all nodes } i} depth_i)/\#nodes$.

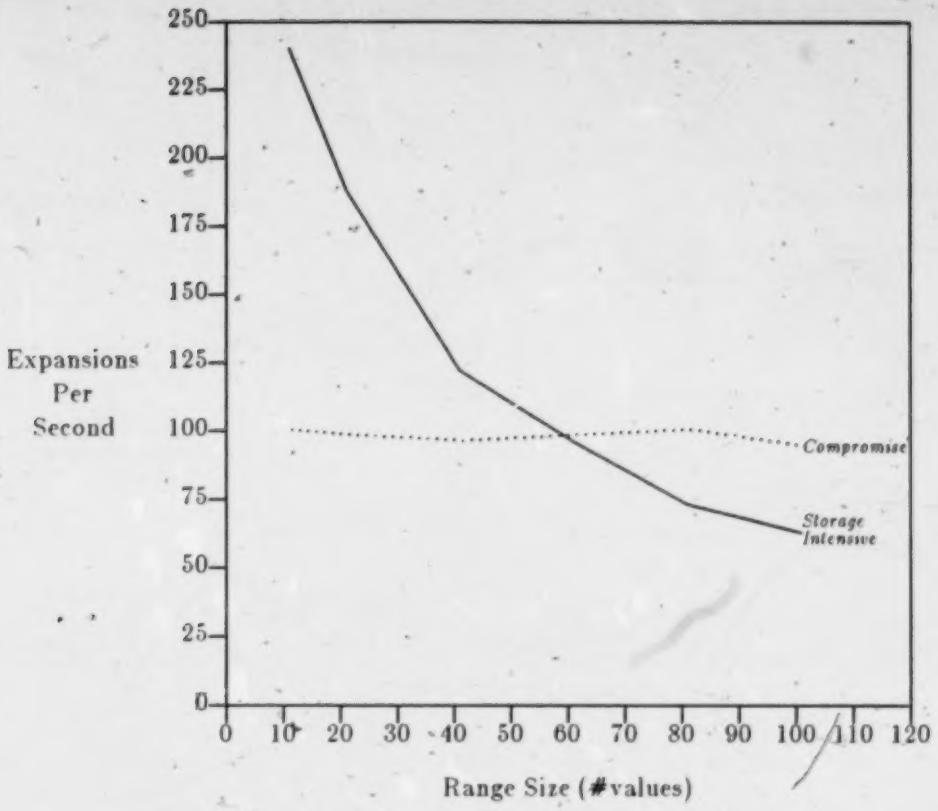

Figure 4.3 Performances for different range sizes.

Figure 4.4 shows the performance comparisons of the two versions for different branching factors. It will obviously take longer to compute conspiracy numbers at parent nodes with a larger number of sons, and hence it is not surprising that there is a performance degradation for larger branching factors. For the experiments that produced the results for figure 4.4, the range size and average depth were deliberately chosen to show that the performances and performance degradations of the two versions can be about equal. A different set of values for these two parameters will affect the performance and performance degradation for each version differently.

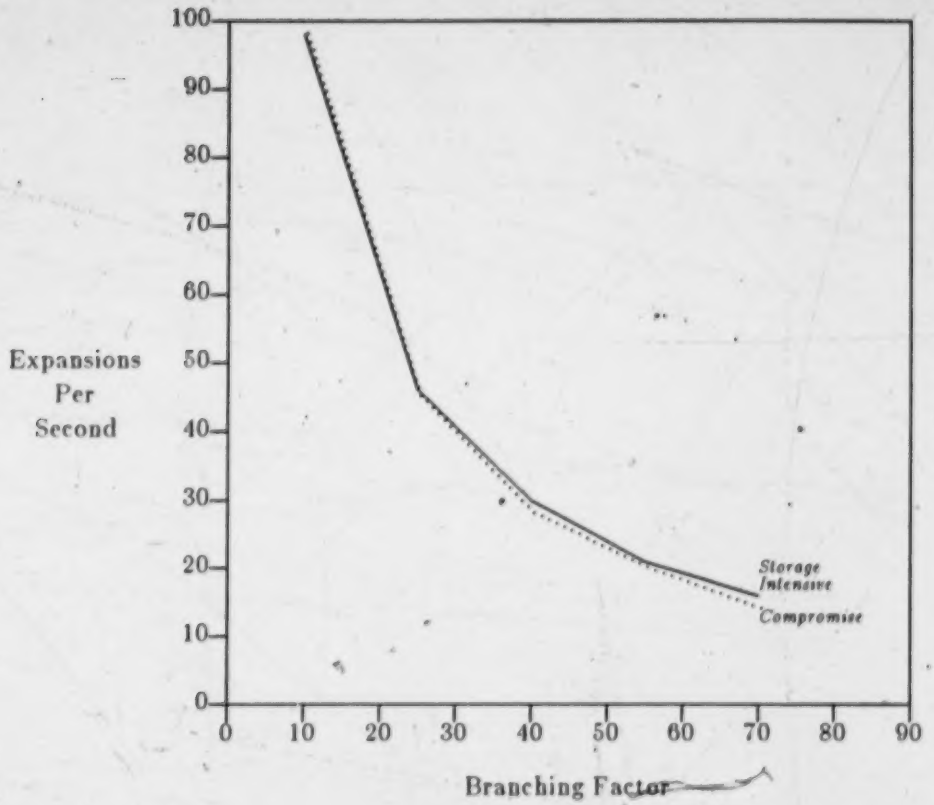

Figure 4.4 Performances for different branching factors.

Figure 4.5 shows the performance comparisons of the two versions for different average depths. It will obviously take longer to update the conspiracy numbers along longer paths (i.e. when the nodes that being expanded are deeper in the tree). Therefore we can expect a performance degradation for trees that have a relatively high average depth, which the results confirm. But note that although the Compromise Implementation's performance is degrading faster in figure 4.5, this situation can be easily reversed by altering the other two parameters (i.e. the branching factor and the range size).

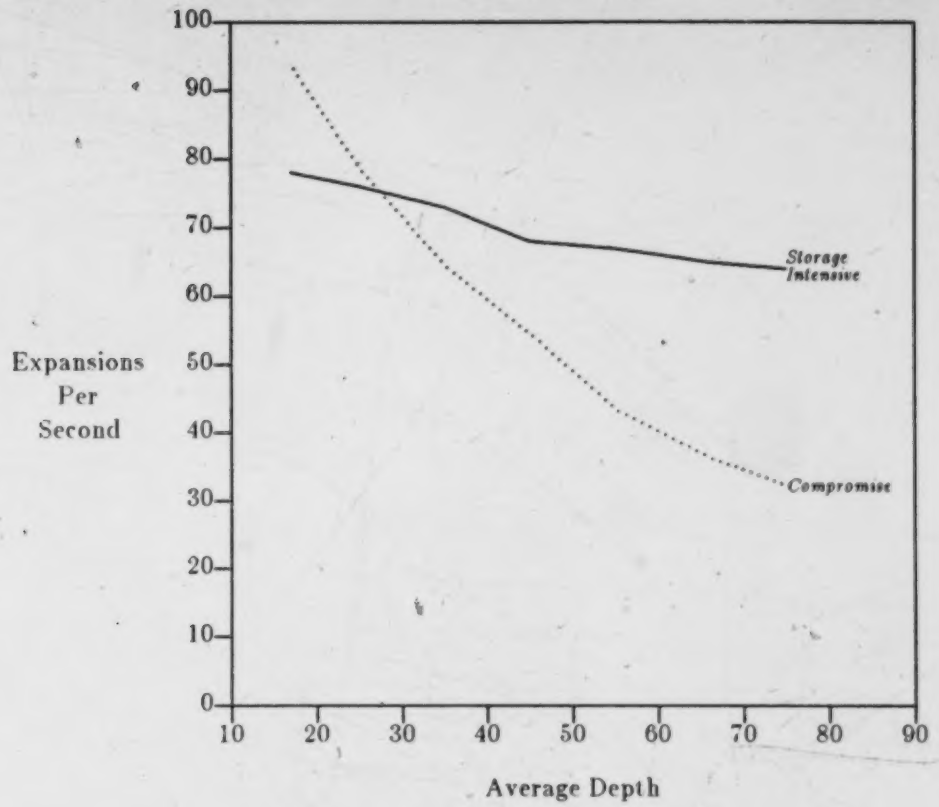

Figure 4.5 Performances for different average depths.

A final matter of interest is the storage requirements of the three versions. Figure 4.6 shows the amount of memory required to store different sized trees for each version. It should be noted here that the storage requirements by each node in the Basic and Compromise versions are fixed, but vary in the Storage Intensive version depending on the type of node (i.e. leaf nodes and interior nodes have different storage requirements because leaf nodes do not have conspiracy numbers stored at them). Two factors that can alter the average memory requirements per node for the Storage Intensive version are the branching factor, which affects the proportion of leaf nodes that there are in a tree, and the range size. It should therefore not be surprising that for a small range size and large branching factor that the Storage Intensive version requires less memory that the Compromise version to store identically sized trees.

Version	Range Size	Branching Factor	Nodes	Memory (bytes)	
				Total	Average
Basic	10	40	10001	80008	8
Storage Int.	10	40	10001	64006	6.4
Compromise	10	40	10001	120012	12
Basic	60	5	10001	80008	8
Storage Int.	60	5	10001	192006	19.2
Compromise	60	5	10001	120012	12

Figure 4.6 Storage Requirements of the three versions.

4.2. General Results

This section summarizes general observations made in our experiments with the Conspiracy Numbers algorithm.

First, a word about the static evaluation function (i.e. the function that assigns a new leaf node its initial value). The observed behavior of the algorithm differs according to the type of static evaluation function used. An initial version of this function assigned totally random node values to new leaf nodes. This however caused situations where a newly expanded node, that had a *good* value before, ended up receiving a *bad* minimax value. In games, situations like this do not occur often (i.e. most minimax values of nodes usually remain fairly stable). So in order to approximate game trees, the leaf node evaluator function was altered so that the node values of the new leaf nodes caused less variation in their parent node's value. This should give us a more accurate idea of how the algorithm will behave in real game situations.

It is clear that the Compromise version's performance is comparable to the Storage Intensive version's performance. The main reason the Compromise version does so well is that *Vmin* and *Vmax* remain stable. Observations of various types of runs showed that *Vmin* and *Vmax* change their value during less than 0.5% of the iterations, and when they do change, the difference between the old and new value is on average about 1.25.[9] This is an indication that the rate at which convergence occurs

[9] This average excludes the relatively large changes that occur at the beginning of the search

will be slow.

Usually when *Vmin* or *Vmax* do change, *Vmin* tends to increase more often than it decreases and *Vmax* tends to decrease more often than it increases. In other words, *Vmin* and *Vmax* move towards each other more often than apart from each other. If this were not the case, convergence on a single root value would be nearly impossible. However, in our experiments, *Vmin* and *Vmax* did diverge about 25% of the time changes to either one occurred, which is a concern because it slows down the rate at which the algorithm converges on a root value.

The rate of convergence is indeed slow. Figure 4.7 shows the average number of expansions it takes to reach total convergence (i.e. *Vmin* = *Vmax*) at several conspiracy thresholds. These results show that total convergence takes longer to achieve at both higher thresholds and for larger range sizes. The main reason for this is that the range of likely root values starts out larger for higher conspiracy thresholds as well as for larger sizes of the range of legal node values. Note that the results in figure 4.7 depend, to a certain degree, on the static evaluation function and how random the values that it assigns to new leaf nodes are.

Range Size	Branching Factor	Conspiracy Threshold	Expansions	Nodes Generated
40	40	2	1265	50600
40	40	3	2289	91560
40	40	4	3804	152160
40	40	5	6330	253200
200	40	2	2569	102760
200	40	3	9145	365800

Figure 4.7 Number of expansions until total convergence occurs.

The Conspiracy Numbers algorithm chooses to rule out *Vmax* on average about the same number of times it chooses to rule out *Vmin*. However, this is an average over many runs. On single runs there is a great variation largely due to two factors:

when the tree is first being grown.

1) that *Vmin* and *Vmax* change slowly and as a result the algorithm chooses the same strategy for many consecutive iterations,

2) and that there are a relatively small number of iterations in a single run.

There are several consequences due to Conspiracy Numbers being a depth-first algorithm without a depth limit. Those that may concern us are:

1) The algorithm explores some branches too deep. For example, many small search trees had maximum depths exceeding 50!

2) The algorithm spends too much time exploring only the first few branches at the root, so that when the search is arbitrarily halted (i.e. before it has converged on a single root value but has run for a reasonable amount of time) many branches at the root remain unexplored.

This may cause some problems in an actual application, such as chess, where time limits exist. By putting too much search effort into the first few branches, other branches are being overlooked. If however a depth limit can be incorporated into the Conspiracy Numbers algorithm, the amount of exploring that is done on these few branches can be limited, resulting in more exploration of some of the other "overlooked" branches.

4.3. Conclusion

In this chapter, we learned about Conspiracy Number's performance and behavior. When choosing an implementation version, we now know that there are only two practical choices: the Storage Intensive version and the Compromise version. Factors that will ultimately influence our choice are: the range size, the branching factor, and the average depth of the tree that will be grown. Probably the most important of these factors to consider is the range size, because it affects mainly the Storage Intensive version whereas the other two factors affect both versions roughly the same.

Of considerable concern is that the rate of convergence is slow. When using the

Conspiracy Numbers algorithm in an application, small range sizes and small conspiracy thresholds should be used to ensure that it has a chance to converge within a reasonable amount of time. Also a concern are the possible problems that may result because Conspiracy Numbers is a depth-first algorithm that functions without a depth limit. Before the algorithm is actually put into an application, it is probably a good idea to first incorporate a depth limit into it to alleviate the problems that will be caused.

Chapter 5

Incorporating Conspiracy Numbers in a Chess Program

The objective in this chapter is to test the conspiracy numbers algorithm in an application. The game of chess was chosen as the candidate application for several reasons: 1) because of its popularity as an application for research in minimax search algorithms, 2) because there are problems in chess that conventional alpha-beta searches can never hope to solve, and 3) because the results from our experiments can be readily compared to the work of others.

A major concern in any application that will use the conspiracy numbers algorithm is speed. The goal of any reasonable chess program is to generate a good response in under three minutes (tournament conditions). From previous experiments on randomly generated trees, we know that the algorithm converges slowly because usually a large number of nodes must be generated. This problem may be compounded by a node evaluator that requires a lot of knowledge and will therefore slow down the rate of node expansions.

Tactical chess is a simpler form of chess which is mainly concerned with the winning and losing of material. A program that solves tactical chess problems requires less knowledge in its leaf node evaluator than a program that can also solve positional chess problems, and hence will be able to generate nodes at a faster rate. In addition, such a program does not require the fine-grained scale of node values needed for positional chess (i.e. a smaller range of node values is sufficient) which means that convergence will be faster. It is for these reasons that our initial program will only solve tactical chess problems.

This chapter consists of two main sections. The first section describes how to implement the Conspiracy Numbers algorithm in a chess program. The second section presents the experimental results.

5.1. Implementing Conspiracy Numbers in Tactical Chess

In our approach to this implementation, there are two parts: an implementation of the conspiracy numbers algorithm, and an implementation of the application dependent data structures and routines. The task of implementing a chess program then becomes one of combining these two parts. To simplify things even further, we will assume that there is no data sharing between these two implementations, and all interactions are done through procedure calls. Since the conspiracy numbers algorithm directs the search, it is actually in control of the total process and calls on the chess implementation part to change its state or provide it with certain information. Schematically, the relationship between these two parts looks something like this:

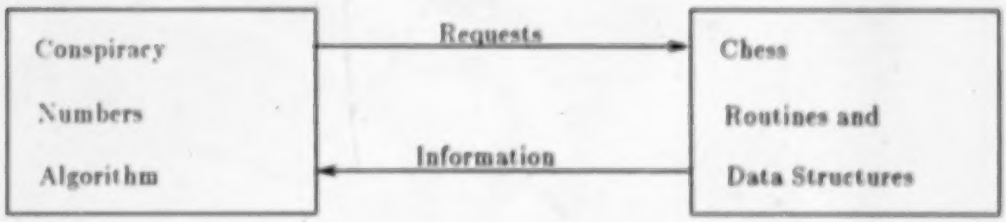

Figure 5.1 Overview of our chess program.

5.1.1. Integrating Chess Knowledge into the Algorithm

Introducing application dependent knowledge into a search algorithm basically involves deciding where and how it will be used, and what information needs to be stored at the nodes. In the conspiracy numbers algorithm, application knowledge is only needed to generate new moves and evaluate them. The generation and evaluation of moves will obviously take place in the routine Expand(). All other phases of the algorithm don't require application dependent knowledge; they are driven primarily by the node values or conspiracy numbers. As for what information needs to be saved at each node, we will only add one new field to store a move.

Storing only the move at each node minimizes the amount of chess information that each node needs to carry. Associated with each node in the search tree is the state information of the game. This state information includes such things as the arrangement of pieces on the board, whose turn it is, the material balance, and many other pieces of information. Such information is maintained strictly by the chess side of the implementation. If we associate an *initial state* with the root node, we can generate the state corresponding to any other node in the tree by making the series of moves stored at each node along the path from the root node to that node. Similarly, we can undo such states and restore the initial state by undoing these moves, in reverse order though.

To expand a node involves interacting with the chess side of the implementation. First, the program generates the game state corresponding to the leaf node that is to be expanded. The Expand() routine then asks the chess side of the program to generate all the moves that can be made from this game state and assigns each move to a new node. The moves are then evaluated by another application dependent routine that takes a single move as a parameter and returns a value. Finally, after all the moves have been evaluated, the initial state is restored.

The interface between the conspiracy numbers algorithm and the chess implementation appears to be simple. It requires four routines and a move representation. The set of routines consist of: make_move(), unmake_move(), generate_moves(), evaluate_position(). The move representation can be handled as an abstract data type, since it is of no concern to the conspiracy numbers algorithm.

5.1.2. Basic Control Mechanism

Another important implementation issue is how the conspiracy numbers algorithm is to be controlled. This involves two things: deciding what conspiracy thresholds to pass the algorithm, and deciding when and how to halt the algorithm. But first it may be useful to review the control mechanism used for a different search algorithm.

The driver routine for the Alpha-Beta search algorithm uses an enhancement called *iterative deepening* [SIA83]. This driver routine repeatedly calls the Alpha-Beta algorithm, each time asking it to build a search tree that is one ply deeper than the previous one. One of the main reasons for using iterative deepening is that when the program runs out of time and has not completed building the current search tree, it can at least use the result from the previous search tree it built.[10] For this search algorithm, the level of confidence one has in the final answer can be gauged by the depth the algorithm managed to search to. So one way of viewing the driver routine's goal is that it tries to increase the level of confidence each time it calls on the search algorithm to grow a tree one level deeper.

A similar idea can be used for the driver routine for the conspiracy numbers algorithm. Here the conspiracy threshold is used to gauge one's confidence in the final answer. Generally, the higher the conspiracy threshold that a problem is solved for, the higher our confidence in the final answer. But remember that a problem is harder to solve for higher conspiracy thresholds. Like the driver routine for the Alpha-Beta search algorithm, this driver will increase the level of confidence in the final answer by repeatedly calling the conspiracy numbers algorithm, each time with a higher conspiracy threshold. Except here, the search algorithm will not build a new search tree each time it is called, but simply extend the tree from the previous call. The

[10] Another reason for using iterative deepening is that it improves the move ordering at the root for each successive call to Alpha-Beta and therefore improves the efficiency of Alpha-Beta.

thresholds used will start at 2, which is the lowest possible, and be incremented by one each time. The main reason for using an iterative increase to the conspiracy threshold is that when the program runs out of time and has not converged for the current threshold, it can at least use the result from the previous completed search.

There are primarily two ways used to halt the algorithm: a time limit, or a node count limit. Using a maximum conspiracy threshold as a consistent criteria to determine when to stop the algorithm is impossible. Assuming that the rate of node expansions is consistent among different problems (i.e. search trees), experiments showed that the number of nodes required to be generated by the algorithm in order to reach a given maximum conspiracy threshold was not consistent among the different problems. Therefore, the conspiracy threshold attained for different problems in a given time limit (or node count limit) will vary.

There are two additional methods of determining when to halt the algorithm, but the conditions for using either one occur less often. Either of these two methods results in the ordinary search time limit being cut short. The first is to use a maximum conspiracy threshold. When the program reaches this threshold, the confidence in the root value is high enough and it is believed that further search will not change that value. Therefore it is reasonable to stop the search after the program has solved the problem at this maximum threshold. The second method is to detect whether one of the players can force a win. Recall that the values LB and UB at leaf nodes are reserved for flagging terminal node (i.e. end of game states where either player looses his king). Whenever the minimax value of the root node is equal to LB or UB, we know that a player can force a series of moves that eventually leads to a game state where he is the winner. Whenever such a situation is detected, further search is unnecessary.

5.1.3. Problems

With the experience obtained through the testing of our original implementation, we were able to identify two major problems. Both of these problems resulted from the depth-first behavior which is characteristic of this algorithm. The conspiracy numbers algorithm always explores one *line of play* (i.e. a single path in the tree) at a time, expanding it deeper and deeper until no more appropriate conspirators can be found there. The two problems this causes are: exploring some lines of play to ridiculous depths, and repetitive game states along a single path throwing the program into an infinite loop.

The danger of permitting the algorithm to search as deep as it wants to is that it may not be able to explore adequately certain parts of the tree in a given amount of time. In one case the algorithm may not have enough time to explore other moves at the root that may actually turn out to be better. In another case the algorithm may concentrate on a line of play that appears promising, but runs out of time to explore all the opponent's alternatives adequately enough along this line of play, leaving open the possibility that this move may actually lead to a win by the opponent. To prevent some lines of play from being searched too deeply, a depth restriction can be employed. Its result is that it puts a bit more breadth into the algorithm, which allows for increased exploration of the other parts of the tree. Note that this type of depth limit is in no way the same as an Alpha-Beta type of depth limit.

The danger of repetitive game states along a single path in the tree is that it causes the program to go into an infinite loop. To show how this can happen, suppose the algorithm is exploring a line of play and a new leaf node is created which represents the same game state as an ancestor node. For convenience, lets call this ancestor node A, and this leaf node B. Suppose there comes a time in the search when node B is explored further. The subtree that will be grown below node B will be identi-

cal to the subtree that was grown below node A. That is, another node whose game state is identical to node A and B will be created, and it too will be eventually explored resulting in another identical subtree being grown, and so on. This cycle will continue, with no convergence possible, and hence we call it an *infinite loop situation* in the algorithm. It is of course possible to *short circuit* this infinite loop using a depth limit; but this does not prevent the wasteful regeneration of these subtrees. Preferably, we would like an application dependent routine that could detect a move that will regenerate a recently seen game state.

But how should the conspiracy numbers algorithm handle these problems? Preferably, we would like to treat such problems in an application independent way within the conspiracy numbers algorithm. Our solution treats nodes that are too deep or represent repetitive game states as terminal nodes, so that the algorithm automatically avoids exploring them any deeper.

To indicate that these nodes must be treated as terminal nodes, a flag field must be added to each node. The setting of this flag will be done by the routine Expand(), where such nodes are first detectable. The only other minor change to the search algorithm occurs in the routine that computes the conspiracy number of a node, where this flag has to be tested first. If the flag is set, the conspiracy number of this node is computed as follows:

$$CN(v) = \begin{cases} 0 & \text{if } v = \textit{node's value} \\ \infty & \textit{otherwise} \end{cases}$$

This will conveniently force the routines that search for nodes to expand to avoid choosing these flagged nodes.

5.1.4. An Enhancement

When the conspiracy numbers algorithm grows a minimax tree, it will always generate all the nodes in the first two ply. To see why this is the case, let's assume that the program is given a middlegame problem to solve. The first node to be expanded is obviously the root node, which generates all the nodes in the first ply. As of this point, $CN(Vmax)$ is equal to one because only one node in the first ply needs to conspire in order to change the root's value to $Vmax$. In fact, $CN(Vmax)$ will remain equal to one as long as there remains an unexpanded node in the first ply and no other single conspirator, which successfully causes the root value to change to $Vmax$, has been found before all nodes in the first ply have been expanded. Since this is the case most often, all nodes in the first two ply will usually be generated before the algorithm even has the chance to converge once.

Normally the algorithm goes through some computation in choosing the nodes that need to be expanded. To produce all the nodes in the first two ply, the algorithm must expand the root node plus all the nodes in the first ply. Instead of having the algorithm *find* these nodes, it is more efficient to expand them before the algorithm actually starts. This enhancement is implemented as a general extension to the conspiracy number algorithm, because it is application independent in nature.

5.2. Experimenting with Conspiracy Numbers in Tactical Chess

With this being the first implementation of the conspiracy numbers algorithm in a chess program, to the best of our knowledge, there are obviously no prior results available. Therefore, there will be three basic objectives to our experiments. The first objective is to compare the performance for such a program to a similar program utilizing a different search algorithm. The second objective is to try to understand the algorithm's behavior, particularly identifying any problems it may have in such an application. The last objective is to examine how several parameters can affect the

performance of this algorithm. Based on these experiments, it should be possible to fairly evaluate the performance of this algorithm for chess.

5.2.1. The Program

Recall that our chess program consists of two parts: an implementation of the conspiracy numbers algorithm, and an implementation of the chess routines and data structures. For the Conspiracy Numbers algorithm, the storage intensive implementation version will be used because it will give us the best performance possible. Routines and data structures for the chess side of the implementation will come from the chess program Phoenix [Sch86], a frequent competitor in the World Computer Chess Championships.

There are several reasons for using parts of Phoenix for the chess side of the implementation. One of them is that the program itself has been run in numerous chess tournaments and therefore is well debugged. Phoenix is also a performance oriented program; it is currently one of the best computer chess programs in the world. But most importantly, by using a large portion of Phoenix in our program means that both programs will be similar, and therefore results from both programs can be compared fairly. In other words, the comparisons can be focused on the two search algorithms employed by these programs (i.e. conspiracy numbers versus Alpha-Beta).

Of particular interest is the *static evaluator function*. To evaluate a new leaf node, this function builds a tree of capture moves to an arbitrary depth using the Alpha-Beta search algorithm. At the root of the tree, it considers checking moves in addition to the capture moves. The results of the search is the expected material balance of the root.

Since the range of values returned by the static evaluator function will usually be different from the range of node values used by our program, each value returned by this function will have to be appropriately scaled so it fits into our program's range.

However, the values LB and UB are restricted to terminal nodes that represent the end of a game (i.e. for game states where one of the players takes the opponent's king). Therefore, the effective range we will be scaling into is $[LB + 1, UB - 1]$.

Our range of node values will be centered around 0 so that $UB = -LB$. In this specification, positive values will indicate that the move will lead to a gain of material for the player whose turn it is at the root node, whereas negative values indicate a loss of material. 0 will obviously indicate equality, where no win or loss of material is expected.

As it turns out, there is no need for a large range of node values. Because there are only a few type of pieces in chess, we require only a limited number of values to indicate the net amount of material a move wins or loses. However, there is a risk in choosing a range that is too small in that we may not be able to distinguish between two moves that have the same value but are otherwise tactically different by a small amount (i.e. if a larger range of values were used, their difference would be more distinguishable).

But how small of a range are we thinking of? A range of 31 values (i.e. UB = 15 = value of a king) is probably the largest we would be considering.[11] Since we will also try many smaller sized ranges, we will choose the storage intensive implementation version of the conspiracy numbers algorithm because it will give use the best possible performance. However, choosing what the appropriate range size should be is a process of striking a balance between being able to generate more nodes per time unit when the range size is smaller and being able to better distinguish between moves when the range size is larger.

[11] In this scheme, the value of a pawn will be 1.

5.2.2. The Test Data

Many experiments with chess programs have used the 300 position test set taken from the book *Win At Chess* [Rei45]. However, the problems in this set are not difficult and is does not take much effort to solve more than 80% of the problems. So in addition to this test set, a set of 95 problems were taken from the *Encyclopedia of Middlegame Combinations* [Inf85]. Many of these problems, on the other hand, are difficult, even for a human!

We will assume that the correct moves have been predetermined for each of these problems. The criteria for correctly solving a problem is that it find the move that wins the most material without, in the case of 2 or more correct moves, having to decide which is positionally best. However, if there are 2 or more correct moves, the program does not have to find all of them in order for the problem to be considered solved.

5.2.3. Initial Results

How well does our program perform? Figure 5.2 illustrates our initial results, comparing the number of problems solved by our program, called Conspire, and Phoenix. These experiments were done on a SUN 3/50 using the problems from the books *Win At Chess (WAC)* and *Encyclopedia of Middlegame Combinations (EMC)*. Admittedly, the results are not impressive, but they do provide some interesting insight in how the conspiracy numbers algorithm behaves in this application.

Win At Chess				
Time (minutes)	Conspire		Phoenix	
	Correct	Max Depth	Correct	Depth
3	157	12.1	236	6.1
10	208	18.4	253	6.9
30	223	20.6	269	7.6
60	231	20.9	272	8.1

Encyclopedia of Middlegame Combinations				
Time (minutes)	Conspire		Phoenix	
	Correct	Max Depth	Correct	Depth
3	15	13.2	43	6.0
10	27	17.5	49	6.9
30	35	20.3	57	7.4
60	38	20.8	58	7.8

Figure 5.2 Number of problems solved.

One interesting statistic is the average initial conspiracy threshold of correctly solved problems. For problems solved in under 60 minutes, this average is about 2.4. On the other hand, the average number of nodes expanded when these problem are finally solved is about 62,000 which indicates that most of these problems are solved early in the search. As figure 5.3 shows, the additional effort in reaching an average of 630,000 expansions results in answers achieving an average threshold of about 11. This illustrates the value of iterative deepening on the conspiracy threshold.

Win At Chess				
Time (minutes)	When Solved		Finish	
	CT	Nodes	CT	Nodes
3	2.1	12,013	5.2	34,129
10	2.2	25,962	7.1	113,816
30	2.3	40,962	9.7	336,135
60	2.4	59,521	11.1	638,935

Encyclopedia of Middlegame Combinations				
Time (minutes)	When Solved		Finish	
	CT	Nodes	CT	Nodes
3	2.0	10,184	5.1	31,749
10	2.3	24,997	6.6	102,121
30	2.5	52,681	8.2	312,605
60	2.5	69,490	9.5	597,044

Figure 5.3 Conspiracy thresholds of solved problems.

Of additional interest is the average conspiracy threshold of incorrectly solved problems over time. Figure 5.4 shows these statistics. These averages are considerably higher than the average conspiracy threshold when the other problems are solved at, which indicates that these incorrectly solved problems are more difficult to solve correctly.

Time (minutes)	WAC	EMC
	CT	CT
3	4.2	3.4
10	5.7	4.6
30	7.9	5.8
60	8.5	6.6

Figure 5.4 Final conspiracy thresholds of unsolved problems

As Figure 5.2 clearly indicates, Phoenix solves more problems than Conspire does by limiting its depth of search to a fraction of what Conspire achieves. Of the 395 problems that both programs were tested on, Phoenix and Conspire could both solve 249 of them. On the other hand, Conspire solved 20 of the problems that Phoenix could not solve, indicating that these problems required more depth than that what

Phoenix was capable of reaching, and Phoenix solved 81 of the problems that Conspire could not solve, indicating that for these problems Conspire does a lot of searching deep in the tree that is unproductive. Notice that after 30 minutes, the average maximum depth levels off at about 20.5. This is because the maximum depth that the algorithm was allowed to search to was 21. Without the depth limit, there may have been a considerably greater wasted effort by Conspire in searching even deeper in the tree.

Finally, two major concerns of the algorithm are its space requirements and rate of node generations. Figure 5.5 compares the space requirements and number of nodes generated over time by Conspire with that of Phoenix. Here Phoenix generates a significantly greater amount of nodes per time unit than Conspire does. However, Conspire does generate fewer nodes in order to correctly solve about the same number of problems, and hence can be judged superior in this respect. In other words, Phoenix expands many nodes that Conspire may find unnecessary to expand. But even if we do significantly speed up Conspire, its memory requirements can easily exceed the memory available to the system because it stores the search tree in memory; Phoenix on the other hand does not explicitly store the search tree but does use fixed sized tables to store information about the successive search trees it builds.

Time (minutes)	Conspire		Phoenix	
	nodes	bytes	nodes	bytes
3	25624	177830	112531	C
10	90969	633370	351314	C
30	291996	2039333	1019293	C
60	556585	3876672	2001220	C

Figure 5.5 Storage requirements for search trees.

5.2.4. Search Behavior

As was noted once before, the algorithm consistently explores one branch until there are no appropriate conspirators below that son. When there are no conspirators below that branch, and there are alternative branches with appropriate conspirators beneath them, then the algorithm explores the left most one of these alternative sons. Often, however, these alternative sons do not get much (if any) chance for consideration of being explored. This results in circumstances where about 80% of the nodes below a certain node can be found below just one son as is illustrated in figure 5.6.[12]

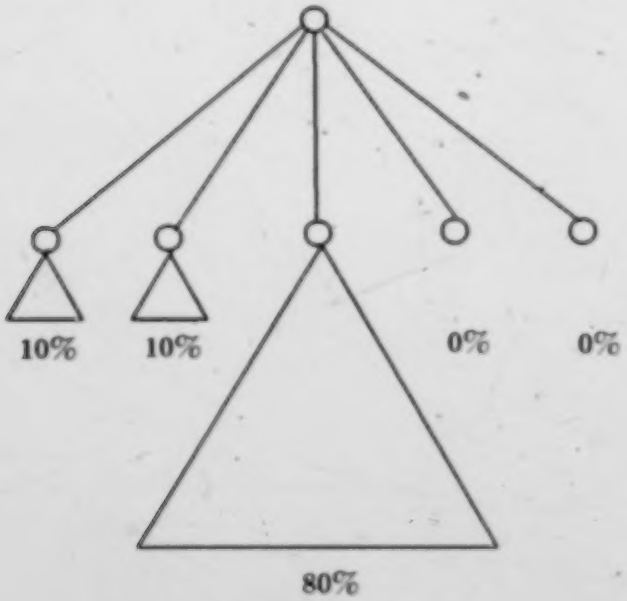

Figure 5.6 Distribution of nodes.

A more detailed examination of how the search effort was being distributed was done on 93 problems. Of the 40 problems that were solved, 37 had the son corresponding to the correct move explored much more often than the other sons. For the remaining 3 problems, the exploration effort was spread more evenly across several sons of the root node and the son corresponding to the correct move was not explored the most.

[12] This is approximately how it looks like beneath nodes on level 0 and 1 of the search tree.

Of the 53 problems that were incorrectly solved, 31 had the son corresponding to the correct move appearing too far down the list, because of a low initial evaluation, which meant that this son had to wait a long time before being considered for exploration. In fact, in the majority of these 31 problems, the correct move was hardly explored at all! In 19 other problems, the correct move was explored much more than the other moves. However, some of the other moves had minimax values that were equal to or higher than the minimax value of the correct move. Here the problem appears to be that the algorithm may not have put any effort into trying to lower the other moves' minimax values. For the remaining 3 problems, the search effort was on moves that appeared further down in the list than the correct move.

There may be two methods we can use to rectify the above problems. The first one is to enhance the static evaluation function so that we can improve the ordering of the sons.[13] A second and more significant improvement would be to modify the algorithm so that it spreads the exploration effort more evenly across the alternative sons (i.e. introduce more breadth at the nodes).

Another problem with this algorithm is that its focus is on finding the most accurate root value and not on finding the best move at the root. This leads to situations where a best move is already obvious, but the algorithm has not converged yet. To see how this can occur, suppose a local *Vmin* and *Vmax* can be computed at each son of the root, using the current conspiracy threshold. If one of these son's local *Vmin* is greater than all other sons' local *Vmax*, this son is clearly the best. In fact, it can be shown that the values for the global *Vmin* and *Vmax* are the same as the values of this son's local *Vmin* and *Vmax*. The consequent of this is that the algorithm will only consider exploring this son, trying to find its *true* value, until it converges. In situations

[13] Alpha-Beta also has an ordering problem. Its efficiency depends on how good the ordering is at the interior nodes. The minimal tree is only possible when the best move is placed first at all interior nodes. In Conspiracy Numbers, ordering may not only determine how efficient the algorithm is, but may also determine whether it is possible for the algorithm to find the solution.

like this, we can either halt the search altogether, or we can increase the conspiracy threshold high enough so that more than a single branch has a chance of being considered and the algorithm can again *separate* out this best move.

5.2.5. Adjusting Parameters

There are primarily two parameters to the algorithm that may affect the number of problems Conspire can solve correctly. The first such parameter is the size of the range of node values. Figure 5.7 shows how many problems are solved when the size of the range is varied. These results are about what was expected. The table shows that too small of a range does not provide enough distinction between moves (i.e. not enough exact information of what is being won or lost) and therefore the number of problems solved is lower. It also shows that increasing the size beyond a particular point does not help solve any more problems. The increase in the time to complete the search as the range size is increased is due to the Storage Intensive implementation version of the conspiracy numbers algorithm that was used. Choosing an appropriate range size finally involves striking an appropriate balance between time and problems solved.

Encyclopedia of Middlegame Combinations				
Range Size	Nodes Expanded			Average Time to Completion (seconds)
	100,000 correct	200,000 correct	300,000 correct	
5	23	27	32	1207
9	27	31	33	1457
13	26	31	35	1677
17	27	33	34	1751
21	29	35	35	1930
25	29	35	35	1987
29	29	35	35	2055

Figure 5.7 Results for different range sizes.

The second parameter is the maximum depth we allow the algorithm to search. Figure 5.8 shows how many problems are solved for a few different maximum depths.

Here the program appears to do better for maximum depths less than 21, which was used for all experiments this far. Since decreasing the maximum depth increases the breadth of the search, it seems to confirm our belief that there is too much depth and not enough breadth in the conspiracy numbers algorithm used for this particular application. However, these results do not show that there is a clear relationship between decreasing maximum depths and an increasing number of problems solved indicating that some problems need more depth to be solved whereas others need more breadth to be solved.

Encyclopedia of Middlegame Combinations				
Maximum Depth	Nodes Expanded			Average Time to Completion (seconds)
	100,000 correct	200,000 correct	300,000 correct	
21	29	35	35	1930
16	31	37	40	1940
15	32	36	37	1976
12	32	37	38	1910

Figure 5.8 Results for different depth limits.

5.2.6. Summary of Results

These initial results are somewhat disappointing. This is mainly due to the wide gap between the number of problems solved by Phoenix and Conspire. But from these experiments we were able to identify some of the problems that the conspiracy numbers algorithm has in this particular application. Problems dealing with the criteria by which leaf nodes are ordered, or the fact that there is too much depth and not enough breadth in the algorithm, or the fact that the focus of the algorithm is too much on finding the right root value, can all be somewhat rectified. These problems give us some direction for further work on this algorithm as well as a hope that at least some improvements to it are possible.

Chapter 6

Improving the Search Strategy

How can we improve the conspiracy number algorithm? The goal is to improve the speed at which convergence occurs, as well as the speed at which the best answer may be *discovered*. This can be done by altering the order in which nodes are examined as well as the set of nodes that are explored over a single run. In other words, we want to change the search strategy without altering the basis of the algorithm. In order to see where and how improvements can be made, we must first re-examine the parts of the algorithm that directly or indirectly influence the set of nodes that it explores.

One way to view the algorithm is that it tests sets of leaf nodes to see if they will conspire to change the root node's value. If the algorithm is testing the possibility of the root node taking on a certain value, it should obviously first test the smallest sets of leaf nodes that could cause such a result (i.e. the minimal conspiracy set) because conspiracies among fewer nodes are more likely than conspiracies among many leaf nodes. The assumption here is that the probabilities of leaf nodes changing their value to any other value are approximately equal. In reality though, we sometimes know which leaf nodes are more likely to change their value appropriately, and hence we could test sets containing these nodes first and hopefully achieve the desired result more quickly. Conspiracy numbers play an essential role in guiding the search procedure down the tree in order to locate the leaf nodes that are members of these minimal conspiracy sets. This forms the basis for the algorithm and cannot be changed.

During each iteration the algorithm must choose one leaf node, which must be a member of a minimal conspiracy set, for expansion. There are parts of the algorithm that influence this particular process, and it is these parts that we want to consider

changing. By changing these parts, we can change the set of nodes that are examined by the algorithm as well as the order in which the nodes are examined.

To find such a leaf node, we must traverse the tree using conspiracy numbers as a guide. At any interior node we traverse through, several sons may have an appropriate leaf node below them. A *rule* is used to choose the order in which alternative branches are considered. McAllester's original rule required that the left most such son be chosen. From initial experiments (Chapter 5), the resulting depth first behavior does not produce good results. Obviously this is the place where we can change such behavior.

The order in which the nodes are placed in also has an effect on the order in which nodes are explored as well as the set of nodes explored. Originally, the nodes were ordered according to their initial value so that the better ones appear first and therefore are explored first. But, in the application of tactical chess, this value does not reflect the potential value of a node if it were explored further so that there cannot be a guarantee that the first node is the best. Maybe if we can use additional knowledge, without affecting the node values, we can place nodes with a better potential first and get better results.

At any one time, the algorithm will only consider choosing leaf nodes from minimal conspirator sets for changing the root value to either *Vmin* or *Vmax*. Since values outside the range [*Vmin*, *Vmax*] are considered unlikely, it makes no sense testing conspirator sets for them. The reason we specifically choose only sets for *Vmin* and *Vmax* is because the algorithm tries to narrow this range gradually by *proving* that the leaf nodes in these minimal conspirator sets do not conspire. But conspirator sets for other values in this range can also be considered, if another approach for narrowing this range is desired.

If we consider using minimal conspirator sets for *Vmin* and *Vmax* only, we should

also identify what influences their value. Both *Vmin* and *Vmax* are a function of the conspiracy threshold (*CT*). Recall that the sequence of values that *CT* can take on is fixed (i.e. $CT = 2,3,...,MaxCT$). Also recall that the value of *CT* is incremented whenever the algorithm converges on a root value (i.e. when *Vmin* = *Vmax*). To change the sequence of values that *Vmin* and *Vmax* take on during a run, we can change the definition of convergence for the algorithm (i.e. allow it to occur earlier) so that we alter the points, in a particular run, when CT is incremented.

For the four sections in the algorithm that were identified as candidates for improvements, examples of alterations that were examined will be presented. The results of applying these alterations will be presented at the end of this chapter. An abbreviated version of this material appears in [KIS88].

6.1. Alternative Rules for Choosing a Branch

How can we improve on the original rule used for choosing among alternative branches? Obviously any new rule should address some of the problems that the old rule may have caused. One such problem is that McAllester's original rule produced an algorithm with a depth-first behavior that had difficulty finding the correct answer quickly. It was suggested that more breadth would spread the exploration effort more evenly in the tree and therefore probably find the solution more quickly. Another problem is that the rule does not result in the *ideal* node being picked for expansion. For example, if it were possible, it would make more sense to first test those minimal conspirator sets that are more likely to cause a particular change in the root value. To a certain degree it would be possible to identify the leaf nodes of such sets if we had a bit more knowledge about the possibility of a leaf node changing its value and how it would change its value.

In an attempt to correct these two problems, several new rules were devised. These rules fall into one of two categories according to whether they require additional

computed information to determine which branch should be descended. Interestingly enough, the rules that introduce more breadth into the algorithm all fall into the category of not requiring additional computed information to make this decision whereas the rules that try to pick the "ideal" node to expand do require computed information in addition to that which is currently provided in the data structures and hence falls into the other category.

6.1.1. Rules Requiring No Additional Computed Information

In order to provide more breadth, we will use a round robin approach for spreading the exploration effort among more of the alternative branches. In McAllester's approach, the search for which branch to descend down always started at the first son and stopped when an appropriate branch was found. When the algorithm traversed through a node many times, this usually resulted in the same son being chosen many times in succession. In a round robin approach, the search always starts at the branch immediately adjacent to the one that was previously chosen and cycles around until an appropriate branch is found. If there are several appropriate branches that can be descended, this approach then gives them each a greater opportunity of being chosen.

To help describe and name the various types of round robin we intend to use, new terminology will be introduced. This new terminology relates to how nodes are viewed by the search routines Rule_out_Vmax and Rule_out_Vmin. Basically, we want to classify a node as being either an OR node or an AND node.

Suppose the goal is to find a leaf node that is a member of a minimal conspiracy set for increasing the root value to Vmax. The routine that traverses the tree in search of such a leaf node (i.e. Rule_out_Vmax) will visit alternating maximizing (MAX) and minimizing (MIN) nodes. At MAX nodes, at most one son has to increase its value to Vmax in order to cause the MAX node to do likewise. In this instance, the algorithm needs to show only that one of these son's value will increase to Vmax, and

that is why we will call MAX nodes OR nodes. At MIN nodes, one or more sons have to increase their value to Vmax in order to cause the MIN node to do likewise. In this instance, the algorithm must prove that all the necessary sons' values do increase to Vmax, and that is why we will call MIN nodes AND nodes.[14] So another way of viewing the traversal is that of visiting alternating OR nodes and AND nodes.

The routine Rule_out_Vmin traverses the tree in search of a leaf node that is a member of a minimal conspirator set for decreasing the root value to Vmin. It views MIN and MAX nodes oppositely; MAX nodes are instead viewed as AND nodes and MIN nodes are viewed as OR nodes, but for the same reasons as before.

Breadth can be added to the search by simply altering the search strategy at OR and AND nodes. One simple modification is to have a round robin approach for selecting among alternative branches to descend at either the OR nodes, or the AND nodes, or both.

At OR nodes, several sons may have the minimum number of conspiracy nodes beneath them. Round robin will change the behavior at OR nodes so that if one of these sons is explored once (i.e. has a leaf node beneath it expanded) and this expanded node fails to "conspire successfully", another son will be chosen to be explored the next time (even if this son still has the minimum number of conspiracy nodes beneath it). By spreading the exploration effort over more sons, we hope that a son that achieves the desired value more quickly than the others will be found, and we hope that on average the number of explorations of OR nodes that it takes to achieve the desired value is reduced.

At AND nodes, all sons that have conspiracy nodes beneath them, that are necessary for achieving a desired value, must be eventually explored. There is no advantage

[14] Alternatively, we could have renamed MAX/MIN nodes as MIN/SUM nodes respectively, according to the operations that were used to compute the conspiracy numbers for changing their value to Vmax.

that can be gained by spreading the exploration effort across these sons, and therefore it would be useless to apply round robin at AND nodes.

In addition, we could apply forms of limited round robin to these nodes. For example, we can do a round robin among alternative branches if they are in the first 25% of the sons. This would spread the exploration effort among only the best sons. Another example is that of exploring all the alternative branches an equal number of times (i.e. apply round robin for only a fixed number of rounds) and then revert to another rule (eg. McAllester's original rule, or a rule limiting the round robin to the first 25% of the branches). This would give each eligible son an equal chance of reducing the number of conspirators it requires to change its value appropriately, and after this the exploration effort may be concentrated on only the best of the sons.

As can be seen, there are many rules that can be made up and tested. The round robin rules that we tested include:

- applying round robin (RR) at the OR nodes and applying McAllester's rule at the AND nodes. We call this version *OR RR*.
- applying round robin at both OR and AND nodes. This represents a breadth first approach to the algorithm. We call this *OR RR + AND RR*.
- applying a partial round robin at OR nodes where only the first 25% of the sons will be considered in the round robin (but if the only alternative branches are not in the first 25%, then McAllister's rule will apply) and applying McAllester's rule at AND nodes. We call this version *OR PRR* (Partial Round Robin).
- applying round robin at OR nodes, but limiting it to one round and then reverting back to McAllester's rule, and applying McAllester's rule at AND nodes. This is interesting because it provides the minimum breadth possible and hence we will call this version *Minimal Breadth*. Also interesting is that as a result of giving each son at least one chance of being explored, many of them will be expanded

much sooner than previously under McAllester's rule.

Since breadth is being added to the algorithm, we do expect an improvement over our original version by all the above versions in the number of problems solved correctly. Of the above versions, the one that may out-perform the others is the OR PRR version because it concentrates more exploration effort on the best sons. The Minimal Breadth version may have the worst performance of the above versions mainly because it has the least amount of breadth.

To implement these round robin rules involves only a few simple modifications to the data structures and search routines. Basically we just need to store an index at each node that indicates which son was most recently explored. In the search routines, the search for which branch to descend down begins at the son adjacent to the most recently explored son (but only at those types of nodes where round robin is to be implemented at). In addition, an extra field may be required to count the number of rounds made through all the sons for those versions that limit the round robin to a fixed number of cycles.

6.1.2. Rules Requiring Additional Computed Information

What is the "ideal" node to expand? Suppose we are interested in finding a leaf node that is a member of a minimal conspiracy set for changing the root value to *Vmax* and suppose there are several such minimal conspiracy sets. If we could compute the probability that each leaf node has of changing its value to *Vmax*, we can then compute the probability that each minimal conspiracy set has of changing the value of the root node to *Vmax*. Our definition of the ideal leaf node to expand is the one that has the highest probability of changing its value appropriately and is a member of the minimal conspiracy set that has the highest probability of changing the root value to *Vmax*.

Why would this be the most ideal leaf node to expand? One way to view this is that if the algorithm is trying to rule out *Vmax*, then it should test the most likely minimal conspiracy sets for changing the root value to *Vmax* first. Another slightly different angle to view this from is to assume the objective is to change the root value to *Vmax*, and to do this the algorithm should explore the leaf nodes of the most likely minimal conspiracy sets for changing the root value to *Vmax* first. For similar reasons we want to choose the leaf node that has the highest probability of changing its value appropriately and is a member of the chosen minimal conspiracy set first.

If there is a function that estimates the probabilities of a leaf node changing its value to each legal node value, it would be simple to compute the probabilities of a minimal conspiracy set being successful at achieving a certain value at the root (i.e. just multiply together the probabilities that the member leaf nodes change their value appropriately). However, there are several reasons why we may not want to work with probabilities. Working with probabilities would mean that we have to use floating point arithmetic and store floating point values at nodes which could be expensive. Also, depending on how much knowledge is put into the function that estimates these probabilities, they are just a guess at best. Therefore, to reduce computing and storage costs, we will use integers and only approximate the meaning and computing of probabilities.

At leaf nodes, the likelihood of a leaf node achieving a certain value will be represented by an integer that is in the range [0,*MaxLike*], where the low end of this range represents a low likelihood of the node achieving that value. To compute the *likelihood* of a set of leaf nodes conspiring successfully to change the root value, we will sum together the probabilities that the member leaf nodes change their value appropriately. Normally, a multiplication operation would be used in such a circumstance, but since we are working with integers we chose to use a summation operation so that the resulting numbers do not become too large.[15] To compare the

[15] The multiplication operation will very quickly produce large numbers, especially in chess where the branching factor is about 40.

likelihood values between sets of different sizes then should be based on the average likelihood value per member.

There are some simple recursive relations that we would like to use for computing likelihood values of a node from the likelihood values of its descendants. In what follows, let m denote the minimax value of a node, v the value we would like to change m to, and j an interior node. Let $\uparrow LIKE(v)$ denote the likelihood of increasing a node's value to v and $\downarrow LIKE(v)$ denote the likelihood of decreasing it to v.

At a leaf node, the computation of the likelihood values is handled by a function. Generally, this function will return a value from the range $[0, MaxLike]$ to indicate how likely it is for the leaf node's value to change to another value by being searched deeper. There are many methods available for computing likelihood values at leaf nodes. An example of one such method will be presented at the end of this section.

At a maximizing interior node, to increase the value to v requires only one son to change its value to v. The number of conspirators to change the node to v is just the minimum number of conspirators to increase one of the sons to v. Since the algorithm is only interested in those sons that have this minimum number, the likelihood of increasing the node to v is simply the maximum likelihood of increasing one of these sons to v. This yields the following relation:

$$\uparrow LIKE_j(v) = \begin{cases} 0 & \text{for all } v \leq m \\ \underset{\text{all sons } k \in K}{MAX} \uparrow LIKE_k(v) & \text{for all } v > m \end{cases}$$

where

$$K = \left\{ k \mid k \in SONS_j \text{ and } \uparrow CN_k(v) = \uparrow CN_j(v) \right\}.$$

To decrease the node's value to v requires all sons whose value is greater than v to decrease their value to v. Given the minimal set of conspirators for decreasing each son to v, all members of each of these sets must conspire together to decrease the

node's value to v. The likelihood of this occurring is simply the sum of the likelihoods of each of these sons decreasing their value to v. Therefore:

$$ {}^{i}LIKE_j(v) = \begin{cases} 0 & \text{for all } v \geq m \\ \sum_{\text{all sons } k \in K} {}^{i}LIKE_k(v) & \text{for all } v < m \end{cases} $$

where

$$ K = \left\{ k \mid k \in SONS_j \text{ and } v < m \right\}. $$

For minimizing interior nodes, the following dual relations apply:

$$ {}^{i}LIKE_j(v) = \begin{cases} 0 & \text{for all } v \leq m \\ \sum_{\text{all sons } k \in K} {}^{i}LIKE_k(v) & \text{for all } v > m \end{cases} $$

where

$$ K = \left\{ k \mid k \in SONS_j \text{ and } v > m \right\}. $$

and

$$ {}^{i}LIKE_j(v) = \begin{cases} 0 & \text{for all } v \geq m \\ \mathrm{MAX}_{\text{all sons } k \in K} {}^{i}LIKE_k(v) & \text{for all } v < m \end{cases} $$

where

$$ K = \left\{ k \mid k \in SONS_j \text{ and } {}^{i}CN_k(v) = {}^{i}CN_j(v) \right\}. $$

How will the search routines use these likelihood values? At OR nodes, where these likelihood values are computed using the MAX operation, the algorithm will traverse down the branch that has the lowest conspiracy number and highest corresponding likelihood value. At AND nodes, where these likelihood values are computed using the summation operation, the algorithm will descend down the branch that has a non-zero conspiracy number and the highest corresponding average likeli-

hood value per conspirator. Note that this procedure will not always find the leaf node with the highest likelihood value if it is a member of a larger conspiracy set. But it should always pick a leaf node with one of the higher likelihood values from such sets. Overall though, this procedure does approximate our objective of finding the *ideal* leaf node to expand. We will call this enhancement simply *Likelihoods*.

Since we are already using the storage intensive implementation version of conspiracy numbers in our chess application, we will use an identical approach for implementing Likelihoods. That is, we will store a likelihood value for each legal node value at all the nodes, and update them in a similar fashion. We will initially allocate two bytes per likelihood value, which may be excessive but allows us to experiment with this enhancement more freely. One byte would be sufficient if *MaxLike* is low enough (eg. *MaxLike* = 8), especially considering that the size of the vast majority of conspiracy sets it considers exploring if fairly small (i.e. recall that from previous experiments the maximum threshold reached was only 26, so with *MaxLike* = 8 the maximum likelihood the algorithm may be looking at when it searches for a leaf node may be 25 x 8 = 200)

The *likelihood estimation function* we will use to compute the likelihoods at a leaf node in our experiments is going to be simple. The values that will be used at the leaf nodes for this will be in the range [0,8]. They will reflect the accuracy of the leaf node's current value so that values further from the current value are assigned lower likelihoods because they are less likely to be achieved quickly by being explored further. Specifically, the following relations were used to compute the likelihood values at leaf nodes:

$$tLIKE(v) = \begin{cases} 8 & \text{if } v = m \\ 7 & \text{if } (v - m) > 0 \text{ and } (v - m) \le 0.1 * Raise \\ 5 & \text{if } (v - m) > 0.1 * Raise \text{ and } (v - m) \le 0.2 * Raise \\ 3 & \text{if } (v - m) > 0.2 * Raise \text{ and } (v - m) \le 0.35 * Raise \\ 1 & \text{if } (v - m) > 0.35 * Raise \text{ and } (v - m) \le 0.5 * Raise \\ 0 & \text{if } (v - m) > 0.5 * Raise \\ 0 & \text{if } v < m \end{cases}$$

and

$$
{}^{i}LIKE_{j}(v) = \begin{cases}
8 & \text{if} \quad m = v \\
7 & \text{if} \quad (m - v) > 0 & \text{and } (m - v) \leq 0.1 * Rsize \\
5 & \text{if} \quad (m - v) > 0.1 * Rsize & \text{and } (m - v) \leq 0.2 * Rsize \\
3 & \text{if} \quad (m - v) > 0.2 * Rsize & \text{and } (m - v) \leq 0.35 * Rsize \\
1 & \text{if} \quad (m - v) > 0.35 * Rsize & \text{and } (m - v) \leq 0.5 * Rsize \\
0 & \text{if} \quad (m - v) > 0.5 * Rsize \\
0 & \text{if} \quad m < v
\end{cases}
$$

where *Rsize* is equal to the size of the range of legal node values. The numbers in these relations should be taken as initial numbers only; they are based strictly on intuition and not on experimentation.

What kind of results can we expect by using this enhancement? We can expect at least an increase in the number of correctly solved problems because of the (hopefully) improved selection criteria. But one thing that is certain is that, unlike the round robin enhancements described earlier, its is difficult to predict how this enhancement will affect the algorithm's behavior.

6.2. An Alternative Definition to Convergence

A problem with this algorithm is that its focus is on finding the most accurate root value and not on finding the best move at the root. This leads to situations where a best move is already obvious, but the algorithm wastes valuable time because it has not converged yet. To understand this problem more, we will first look at how such a situation can be detected and why it is felt exploration effort is being wasted when this occurs.

Suppose that a local *Vmin* and *Vmax* can be computed for each son of the root node using the current conspiracy threshold (CT). Assuming that one of these son's local *Vmin* is greater than all the other sons' local *Vmax* (i.e. this son's *pessimistic* value is greater than the *optimistic* value of all other sons), clearly this son would be the best. Assume that we have such a situation, where one son is clearly the best, and

the algorithm has not converged yet (i.e. $Vmin \neq Vmax$). What would the behavior of the algorithm be like at such a time?

To understand what the algorithm will try to accomplish, we will first try to find the relationship between the best son's local *Vmin/Vmax* and the global *Vmin/Vmax*. For convenience, let us call the best son's *Vmin* *best_Vmin* and the best son's *Vmax* *best_Vmax*. Recall that *Vmax* is the largest value whose conspiracy number is less than CT and the conspiracy number for increasing a maximizing node is computed using the MIN operation. We know that at the best son the conspiracy number for *best_Vmax* is less than CT and that at all other sons the conspiracy numbers for *best_Vmax* is greater than or equal to CT. In addition, the conspiracy numbers for *best_Vmax* + 1 at all sons is greater than or equal to CT. Therefore we have at the root that $CN(best_Vmax) < CT$ and $CN(best_Vmax + 1) \geq CT$, and hence $Vmax = best_Vmax$. *Vmin*, on the other hand is the smallest value whose conspiracy number is less than CT and the operation used for computing it at the root node is the SUM operation. We know that at the best son the conspiracy number for *best_Vmin* is less than CT and that at all other sons the conspiracy number for *best_Vmin* is 0 (because all their node values are less than *best_Vmin*). In addition, the conspiracy number for *best_Vmin* − 1 obviously is greater than or equal to CT at the best son. Therefore, we have at the root that $CN(best_Vmin) < CT$ and $CN(best_Vmin - 1) \geq CT$, and hence $Vmin = best_Vmin$.

How will this situation affect the search routines that find leaf nodes to expand? Clearly the minimal conspiracy sets for the values in the range $[Vmin, Vmax]$ will all appear under the best son. The result is that the search routines will only pick leaf nodes from under the best son to expand. The best son will continue to be explored until the algorithm converges, at which point we do not only have $Vmin = Vmax$ but also $best_Vmin = best_Vmax$. In other words, the focus of the algorithm has been shifted to finding an accurate value for the best son, or the best move that the

opponent can make if we made the initial move.[16]

It is probably wasteful exploring the best son any further. Instead it may be better to try to prove that this is the best son for a higher conspiracy threshold. At a higher conspiracy threshold, the range of likely node values of some of the other sons may overlap the range of likely node values of the best son, and this would mean that other sons must eventually be considered for exploration.

So what changes to the algorithm are necessary? What we will propose to do is extend the definition of convergence. Currently convergence is defined as occurring when *Vmin* becomes equal to *Vmax*. We will loosen this condition by allowing convergence also to occur when one son at the root becomes clearly better than than the other sons. If convergence occurs because one son is clearly better than the other sons, but *Vmin* does not yet equal *Vmax*, the top level routine will return control to the basic control routine which in turn will increment the conspiracy threshold, and this is exactly what we want to accomplish under such a circumstance.

To implement this is simple. The algorithm needs to only keep track of the current local *Vmax* for each of the sons of the root node. To do this, the algorithm computes the local *Vmax* for each son before entering the main loop of the top level routine, and updates the local *Vmax* corresponding to the son that was explored after each iteration. To check if one son is clearly the best, the algorithm needs to only count the number of local *Vmax's* greater than *Vmin*, and if this count is equal to one then the condition is true because *Vmin* = *best_Vmin* and *best_Vmax* > *best_Vmin*.

We will call this enhancement *Best Move Cutoffs*. But what kind of results can we expect from it? First, the goal of this enhancement is to find the best son earlier and try to "reprove" it at higher conspiracy thresholds. Therefore we expect that problems

[16] Note that it is possible for Vmin and Vmax to diverge. This could lead to a state where other sons may be considered for exploration. However, the probability of Vmin and Vmax diverging is small.

will on average be solved to a higher threshold. But if there will be an increased number of correctly solved problems is not obvious. One could imagine that if less effort is wasted on trying to find the accurate value of the best son that the likelihood of finding a better son, which may turn out to be the "correct" one, may increase. Therefore, we can hope for a marginal improvement in the number of problems solved correctly.

Note that now the algorithm is in effect using a range of likely node values at each of the root node's sons to detect which son is clearly the best. This feature makes it similar to the B* search algorithm. Still the overall procedure used to grow the search trees are entirely different, but we have moved one step closer towards a hybrid of the two algorithms.

6.3. Considering Conspiracy Set for Alternative Values

Currently the algorithm only considers choosing leaf nodes from minimal conspiracy sets for changing the root value to either *Vmin* or *Vmax*. Among the set of conspiracy numbers corresponding to the values in the range [*Vmin*, *Vmax*], the largest one corresponds to either *Vmin* or *Vmax*. Since the conspiracy number for *Vmin* and *Vmax* are also the closest to the conspiracy threshold, these values represent the most likely to rule out. Therefore the objective of the current procedure is to narrow the range of likely root values by trying to show that the values which represent the end points of this range are unlikely to occur.

We can change this general procedure by altering the overall objective. Instead of trying to rule out a particular value, let's have the algorithm try to more successfully change the root value. The most likely value to change the root value to is either *Vroot* + 1 or *Vroot* − 1, whichever has the lowest corresponding conspiracy number. To implement this change is simple because it only involves altering the condition that determines which search routine to call and also altering the parameters to these

routines. In other words, the altered portion will (approximately) appear as:

```
if CN(Vroot−1) ≤ CN(Vroot+1)
    Rule_out_Vmin (Vroot-1);
else
    Rule_out_Vmax (Vroot+1);
```

We will call this alteration *Minimum Windows* because it is analogous to minimal window searches of minimax trees [MRS87].

As for the results, there should be more frequent changes in the root value because of this alteration. But what is not at all clear is the effect that this may have on the number of problems it correctly solves.

6.4. Ordering the New Leaf Nodes

One problem we currently have in our program is that there may not be enough distinction between nodes due to the small range of node values we use. The result of this is that when it comes to ordering new leaf nodes based on these values, all what is occurring is that they are being grouped together according to their value. In other words, there is limited ordering of nodes within these groups themselves.

To provide some type of ordering within these groups, we will assign a secondary value to each node on which to base this ordering on. Application dependent knowledge will be required to compute a secondary value. So in chess, for example, we could give each move bonus points for such things as moving in a forward direction, capturing a piece, or putting the opposing king in check. These bonus points will be stored in the secondary value. It is hoped that this will overall improve the ordering of the nodes, by putting the better ones in front, and hence should lead to a marginal improvement in the results.

6.5. Results

All enhancements presented in this chapter were applied to the Conspiracy Numbers algorithm in the program Conspire (see chapter 5). Experiments were done using the smaller set of problems taken from the *Encyclopedia of Middlegame Combinations*. The programs, each representing a different version of the algorithm, were run on each problem until they had generated a search tree of 300,000 nodes (approximately 30 minutes on a SUN 3/50). The emphasis in the analysis of the results from these experiments is on the number of problems each program correctly solved.

Figure 6.1 illustrates the performance of two programs that differ only in the way they order new leaf nodes. The first program, which is the original Conspire, employs the original ordering method which is based completely on the sons' values. The second program employs the enhanced ordering method described earlier in this chapter. Although the difference in performances is insignificant, these results do suggest that our enhanced ordering method is inferior to the original. More importantly, these results show that it is difficult to estimate which move is better, without more extensive knowledge, when there is no difference in the evaluation of the resulting board positions and that the algorithm should perhaps rely to a lesser extent on the ordering of nodes.

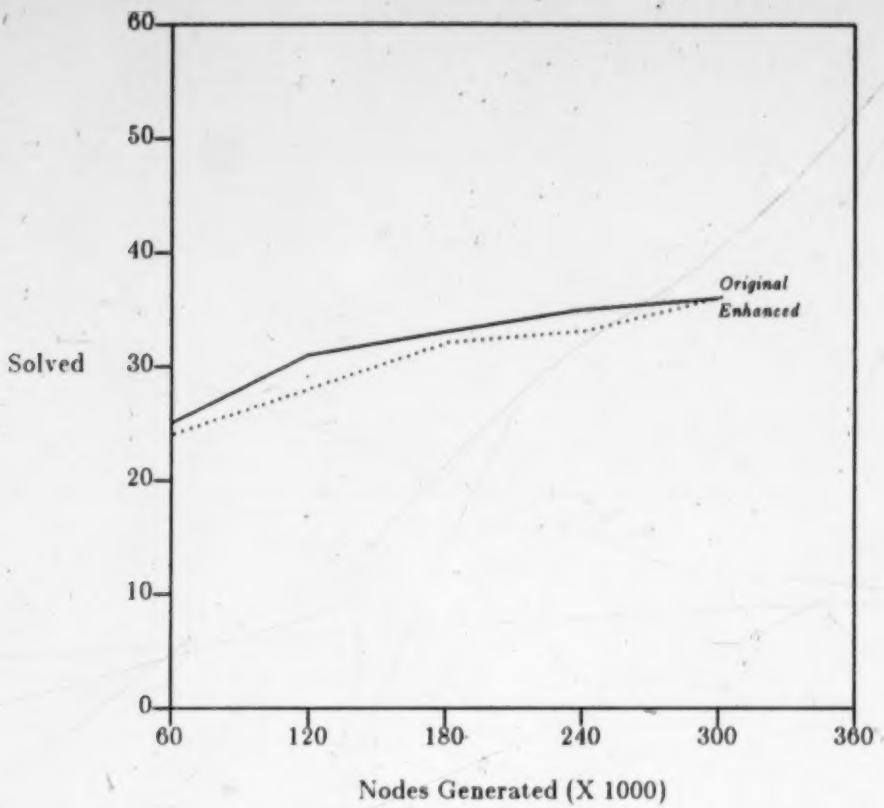

Figure 6.1 Results for different ordering methods.

Figure 6.2 illustrates the performance of the four round robin variants of the Conspiracy Numbers algorithm along with the performance of the original algorithm proposed by McAllester. The results clearly show the advantage of adding some breadth to the search and that the round robin versions all have comparable performances. The round robin versions here gave more sons a better chance of being explored, thus increasing the chances of finding the best move more quickly.[17]

In comparing the problems solved by each, all algorithms were able to solve 32 common problems while 43 were not solvable by any. Of the remaining 20, 1 required the depth possible only through McAllester's version, and 4 required the breadth of the

[17] Experiments with the Minimal Breadth version were also done using the problem set taken from the book *Win at Chess*. The results of these experiments are reported in [Sch88].

round robin schemes. Among only the round robin versions, 39 problems were solvable by all while 44 were not solvable by any.

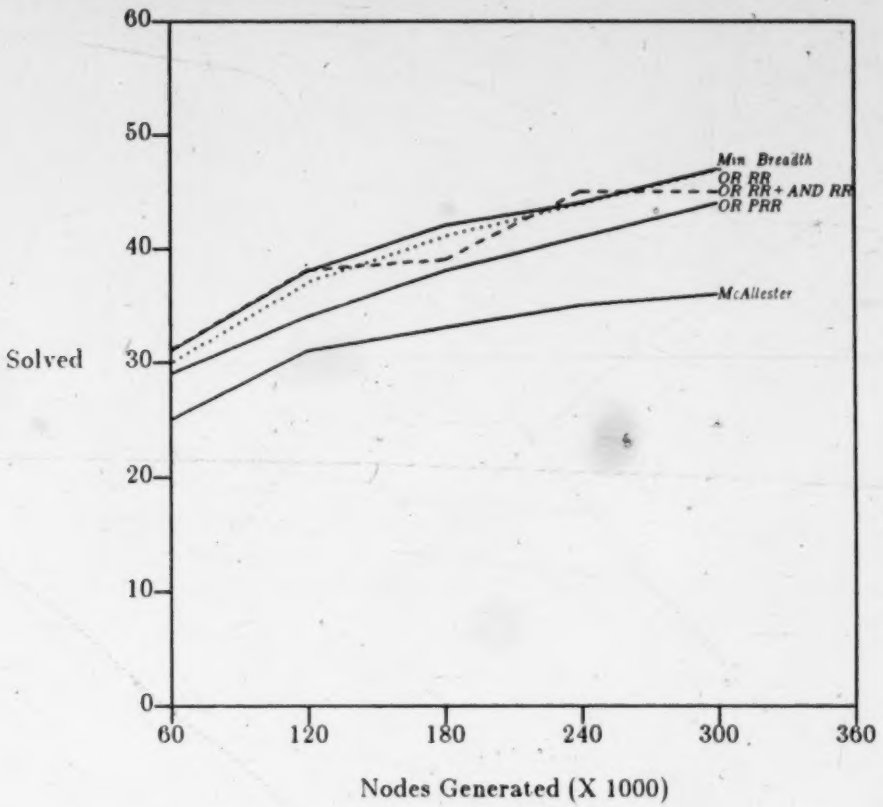

Figure 6.2 Results for the round robin version.

Figure 6.3 illustrates the performance of a program using the *likelihood* version of the Conspiracy Numbers algorithm. These results show that there is a small, but consistent improvement in the number of correctly solved problems over a program using the original version of the algorithm. The problem with the likelihood version appears to be that it has too much of a depth-first behavior: it always chooses to descend down to the left-most son that has the appropriate conspirators beneath it as well as the maximum likelihood for causing the desired result.

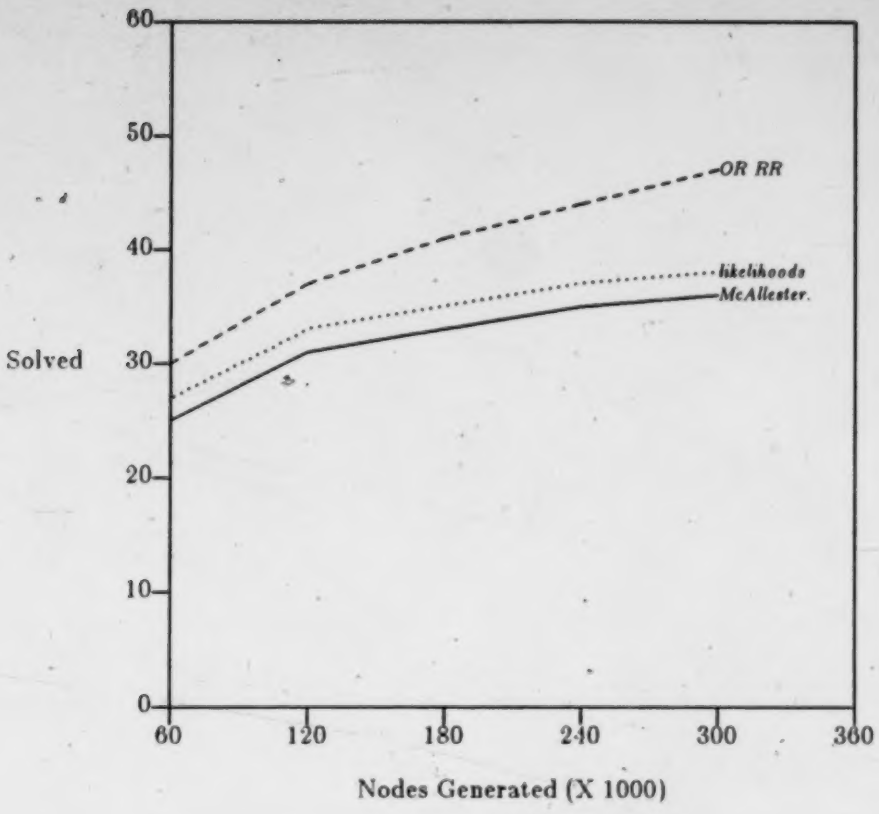

Figure 6.3 Initial results for the likelihood version.

Two simple enhancements, which add more breadth, were made to this likelihood version. The first enhancement involved applying a round robin technique for choosing which branch to descend down. We will call this new version simply *likelihoods + RR*. The second enhancement involved altering the likelihood estimation function at leaf nodes so that more nodes have a better chance of being explored. Specifically, the relations that are used to compute the likelihood values at leaf nodes will be:

$$\uparrow LIKE(v) = \begin{cases} 2 & \text{if} \quad v = m \\ 1 & \text{if} \quad (v - m) > 0 \quad \text{and} \quad (v - m) \leq 0.35 * Rsize \\ 0 & \text{if} \quad (v - m) > 0.35 * Rsize \\ 0 & \text{if} \quad v < m \end{cases}$$

and

$$LIKE_j(v) = \begin{cases} 2 & \text{if} \quad m = v \\ 1 & \text{if} \quad (m - v) > 0 \quad \text{and} \quad (m - v) \leq 0.35 * Raize \\ 0 & \text{if} \quad (m - v) > 0.35 * Raize \\ 0 & \text{if} \quad m < v \end{cases}$$

where *Raize* is equal to the size of the range of legal node values and m is equal to the node's current minimax value. We will call this new version *likelihoods' + RR*.

Figure 6.4 illustrates the performance of these likelihood variants. These results clearly show that adding breadth allows the algorithm to solve more problems. However, we were only able to raise the algorithm's performance to that of the round robin versions. But at what cost do we achieve this? The additional overhead in using and computing likelihood values resulted in an increase in the average amount of time required to search such a tree by 36%. For comparison purposes, the round robin techniques increased the average time by less than 2%. Certainly the additional costs of using likelihood values is not worth the increased performance we were able to get through it.

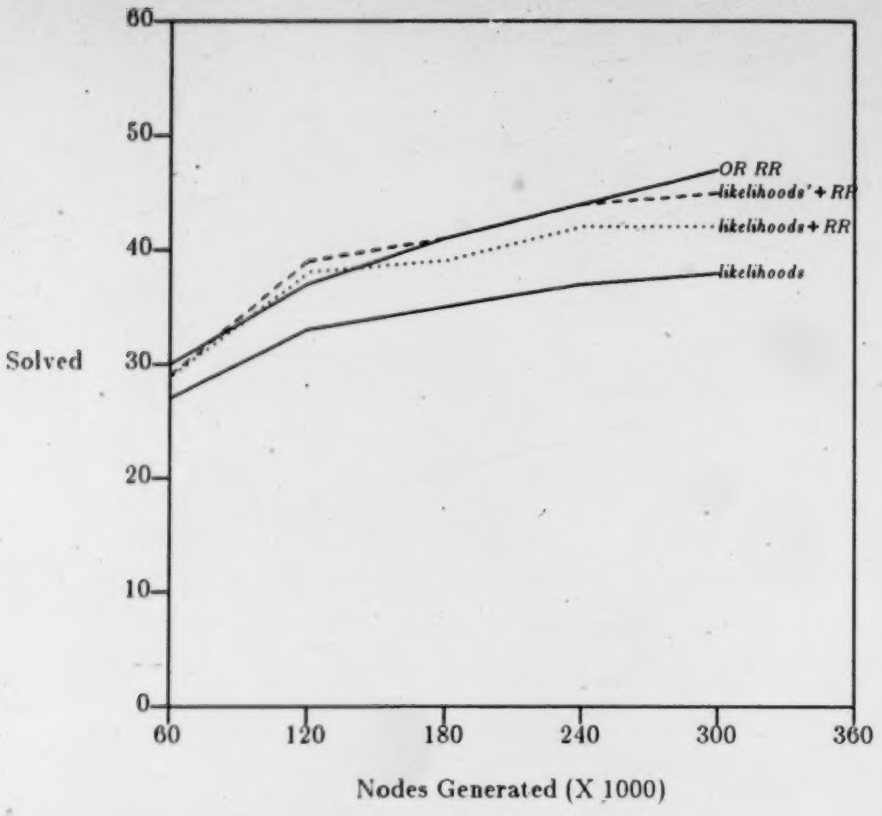

Figure 6.4 Results for variants of the likelihood version.

The *best move cutoff* enhancement was added to a program that was already using the *OR RR* version of the Conspiracy Numbers algorithm. Figure 6.5 compares the performance of the program before and after this enhancement was added. With the enhancement, the program was able to solve 4 new problems in addition to the 47 problems it could solve before. On the 47 original problems, the cutoff was applied an average of 3.7 times per problem causing an increase in the average attained conspiracy threshold from 9.1 to 9.6. On the 4 new problems, the cutoff was applied an average of 2.25 times resulting in the program finding their correct solutions quicker. Overall, this enhancement can be viewed as a significant improvement to the algorithm.

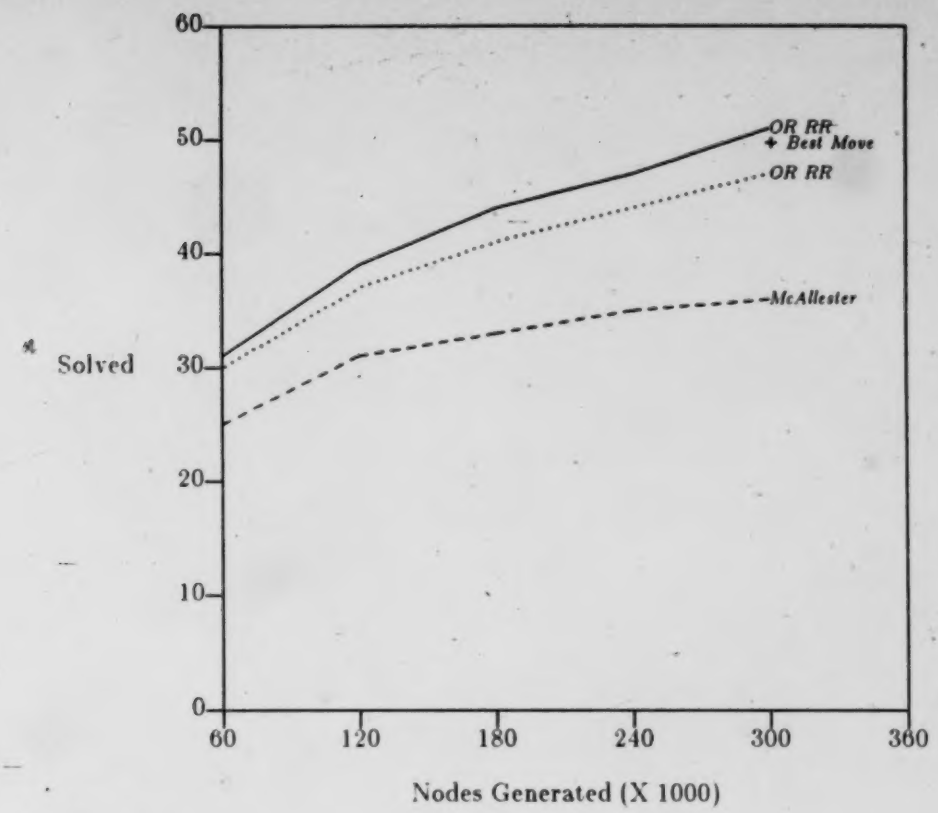

Figure 6.5 Results for the Best Move Cutoff enhancement.

The *minimum windows* enhancement was added to a program that was already using the *minimal breadth* version of the Conspiracy Numbers algorithm. Figure 6.6 compares the performance of the program before and after this enhancement was added. The minimum window enhancement definitely has a negative impact on the algorithm's performance. One reason for the decrease may be that the algorithm is never trying to find the highest root value achievable. For example, when the original algorithm is trying to rule out *Vmax*, it is also testing if *Vmax* is achievable. In order to achieve *Vmax*, one of the root's moves (i.e. sons) must also achieve a value of *Vmax*. So by putting more emphasis on trying to achieve the highest plausible root value, it is actually trying to find the "best" move possible, rather than just a slightly "better" move.

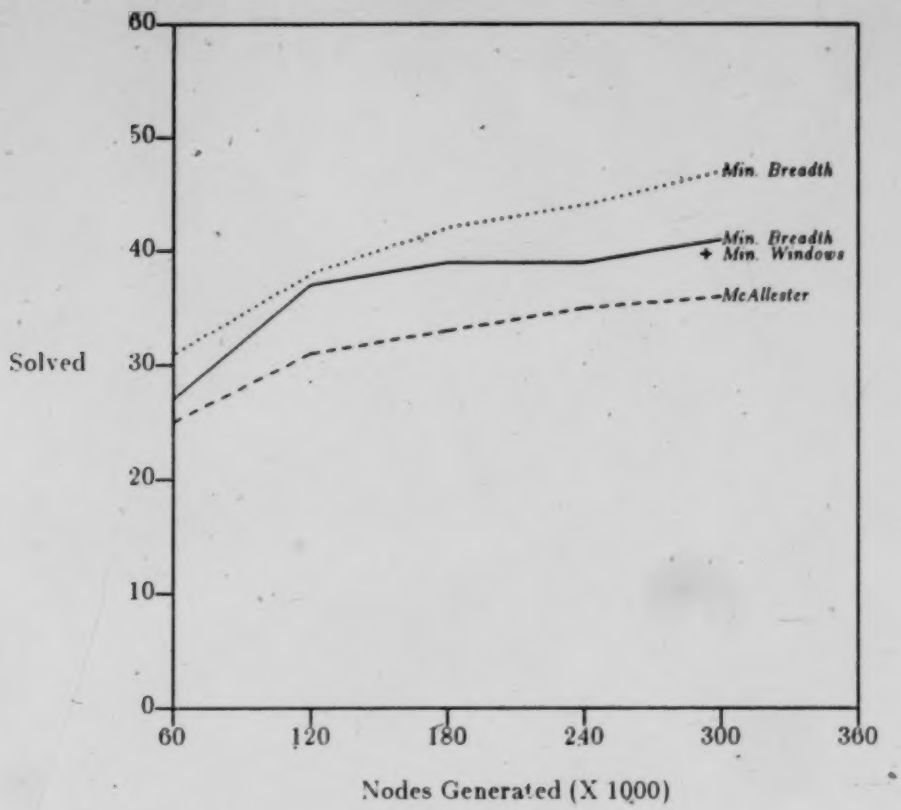

Figure 6.6 Results for the Minimum Windows enhancement.

6.6. Conclusion

This chapter has shown that significant improvements to the Conspiracy Numbers algorithm can be made. The two main improvements that were discussed here were to add more breadth to the algorithm, through round robin techniques, and to increment the conspiracy threshold immediately after a move is found to be clearly better than the others instead of waiting until convergence occurs. Experiments showed that with these improvements, the algorithm was capable of solving up to 41% more problems than McAllester's original proposal. This chapter also showed that we were less successful in using likelihood values and not at all successful in using minimum windows to improve the search.

Chapter 7

Conclusions and Further Work

Conspiracy Numbers is an exciting new approach to minimax search. It has several advantages over conventional Alpha-Beta search approaches, notably the ability to grow trees to variable depth in an application independent manner without any enhancements to the static evaluation function. However, its disadvantages include the question of algorithm convergence (it is not guaranteed to converge) and the problem of determining a satisfactory termination threshold.

There were three major topics discussed in this thesis. The first topic was an examination of the various ways to implement the Conspiracy Numbers algorithm as well as an examination of their relative performances. The second topic was an examination of the algorithms performance in a program that solves tactical chess problems. The last topic was an examination of several alterations to the algorithm where the goal was to improve its performance in chess.

7.1. Results

The main issue involved in implementing the algorithm is the space-time trade-off. To investigate this trade-off, three implementation version were tested. These versions can be distinguished by the number of conspiracy numbers they store at nodes: the Basic version stores no conspiracy numbers at nodes, the Compromise version stores 4, and the Storage Intensive version stores 1 for each legal node value. It was shown that storing conspiracy numbers at nodes significantly improved the algorithm's performance because it reduced the large amount of time spent recomputing them. Hence, the only practical choices turn out to be the Storage Intensive version and the Compromise version. Although several factors affect these two versions' performance, the most important factor to consider when choosing one is the range size; for small ranges, the Storage Intensive version usually outperforms the Compromise version,

whereas for larger ranges the opposite is true.

When the algorithm was implemented in a program that solves tactical chess problems, it was found to perform not as well as a similar program using the Alpha-Beta search algorithm. Several problems were identified here which included: the criteria by which leaf nodes are ordered, that there is too much depth and not enough breadth in the algorithm, and that the focus of the algorithm is too much on finding the right root value instead of the best move.

It was also shown that significant improvements to the algorithm can be made. These improvements included adding more breadth to the algorithm, through round robin techniques, and incrementing the conspiracy threshold immediately after a move is found to be clearly better than the others (i.e. Best Move Cutoffs). Experiments showed that with these improvements the algorithm was capable of solving 41% more problems. Less successful was the use of likelihood values; compared to the performance of other improvements, the increased number of problems solved using this enhancement does not appear to justify the large overhead in computing these values.

7.2. Further Work

The Conspiracy Numbers algorithm is still in its infancy and there is plenty of room for enhancements. In the area of search strategies, a hybrid algorithm combining both B* and Conspiracy Numbers looks promising. The B* algorithm, with its notion of optimistic and pessimistic values, is an elegant proof procedure. A serious defect with it has been the use of artificially constructed probabilities to guide the search. However, conspiracy numbers provide a means of generating these probabilities directly from the search tree.

In a related topic, work can also be done in the area of designing and implementing parallel versions of this algorithm. Since all strategies that were discussed in this thesis are sequential in nature, new strategies that are more suitable for a parallel

version must be created. One interesting example of a search strategy that may work well here is another B*-Conspiracy Numbers hybrid. In this strategy, each move from the root state would be assigned to a single Exploration processor. A Master process would then be responsible for choosing the local strategies, such as Rule_out_local_Vmax or Rule_out_local_Vmin, for each Exploration process. Such a version would help alleviate the problem that some sons of the root were not being explored adequately enough and also put the focus of the algorithm on "separating" out a best move.

And finally, results were disappointing when Conspiracy Numbers was used in a chess program that plays positional chess as well as tactical chess [Sch88]. It was suggested there that the cause of the problem was with the stability of the static evaluation function and that this results in the root having difficulty converging to a value. However, experiments on randomly generated trees do not *appear* to support this view. The speed at which convergence occurs at seems to be more a function of the range size than the stability of the static evaluation function. Since such a program requires a larger range size, due to the fine-grained scale of node values needed, it will experience slower convergence. Overall though, the exact cause is not definite yet, but if Conspiracy Numbers is to be a viable alternative to Alpha-Beta for chess, this problem must be solved.

References

[Ber79] H.J. Berliner, The B* Tree Search Algorithm: A Best First Proof Procedure, *Artificial Intelligence 12*, 1 (1979), 23-40.

[Inf85] Chess Informator, ed., *Encyclopedia of Middlegame Combinations*, Sahovski Informator, Beograd, 1985.

[KlS88] N. Klingbeil and J. Schaeffer, Search Strategies for Conspiracy Numbers, *Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, 1988, 133-139.

[KnM75] D.E. Knuth and R.W. Moore, An Analysis of Alpha-Beta Pruning, *Artificial Intelligence 6*, (1975), 293-326.

[Kor85] Richard Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence 27*, (1985), 97-109.

[MRS87] T.A. Marsland, A. Reinefeld and J. Schaeffer, Low Overhead Alternatives to SSS*, *Artificial Intelligence 31*, 1 (1987), 185-199.

[McA85] D.A. McAllester, A New Procedure for Growing Mini-Max Trees, Technical Report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1985.

[McA88] D.A. McAllester, Conspiracy Numbers for Min-Max Search, *Artificial Intelligence 35*, 3 (1988), 287-310.

[Nau79] D.S. Nau, Quality of Decision Versus Depth of Search on Game Trees, Ph.D. thesis, Dept. of Computer Science, Duke University, 1979.

[Pal82] A.J. Palay, The B* Tree Search Algorithm - New Results, *Artificial Intelligence 19*, 2 (1982), 145-163.

[Rei45] F. Reinfeld, *Win At Chess*, Dover Books, 1945.

[Sch86] J. Schaeffer, *Experiments in Search and Knowledge*, Ph.D. thesis, Department of Computer Science, University of Waterloo, 1986.

[Sch88] J. Schaeffer, Conspiracy Numbers, *Artificial Intelligence*, 1988. In press. Also to appear in *Advances in Computer Chess V*, D. Beal and H. Berliner (ed.), Elsevier Press, 1988.

[SlA83] D.J. Slate and L.R. Atkin, Chess 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, FreyP.W. (ed.), Springer-Verlag, New York, second edition 1983, 82-118.

Appendix 1

The Basic Implementation Version

The following is a pseudo-code description of the Basic Implementation version of the Conspiracy Numbers algorithm. The main feature of this version is that it does not store any conspiracy numbers at the nodes of the tree and therefore has to traverse an entire search tree in order to compute a conspiracy number.

```
/*-------------------------------------------------------------------------*/
/* Some constants used by the program                                      */
/*-------------------------------------------------------------------------*/

#define LB  -40                         /* lower bound on node values   */
#define UB  40                          /* upper bound on node values   */

#define MAX   0                         /* for indicating type of node  */
#define MIN   1

#define DECREASE   0                    /* for purpose of computing CN's*/
#define INCREASE   1                    /* indicates direction of change*/

#define INFINITY   255                  /* maximum conspiracy number    */

/*-------------------------------------------------------------------------*/
/* Info stored at all nodes of the tree                                    */
/*-------------------------------------------------------------------------*/

struct Node {
        short int value;                /* minimax value of this node   */
        short int numSons;              /* number of sons               */
        struct Node *sons;              /* pointer to array of sons     */
};

/*-------------------------------------------------------------------------*/
/* Structure for storing a path between two nodes                          */
/*-------------------------------------------------------------------------*/

struct Node *SearchPath[ MAXDEPTH ];
int PathDepth;                          /* index of last node in path   */

struct Node *RootNode;                  /* pointer to root node         */

/*-------------------------------------------------------------------------*/
/* Main routine for the Conspiracy Numbers Algorithm. Takes a              */
/* conspiracy threshold as parameter.                                      */
```

```
/*-------------------------------------------------------------------------*/
Top_Level( CT )
{
    if ( RootNode == NULL )
        Initialize_root();

    Vroot = RootNode->value;
    Vmin = Compute_Vmin( Vroot, CT );
    Vmax = Compute_Vmax( Vroot, CT );

    while ( Vmin != Vmax ) {

        /* find a path to a node to expand      */
        if ( ( Vmax - Vroot ) > ( Vroot - Vmin ) )
            aNode = Rule_out_Vmax( Vmax );
        else
            aNode = Rule_out_Vmin( Vmin );

        /* descend to the node and expand it    */
        Expand( aNode, PathDepth );

        /* recompute the minimax values along   */
        /* path. Also recompute Vmin and Vmax.  */
        Update_minimax_values();
        Vmin = Compute_Vmin( Vmin, CT );
        Vmax = Compute_Vmax( Vmax, CT );
        Vroot = RootNode->value;
    }

    return( Vroot );
}

/*-------------------------------------------------------------------------*/
/* Computes a new value for Vmax. Takes the old Vmax and the conspiracy */
/* threshold as parameters.                                             */
/*-------------------------------------------------------------------------*/
Compute_Vmax( Vmax, CT )
{
    /* use old Vmax as a starting point.            */
    /* make sure Vmax is >= to current root value   */
    if ( RootNode->value > Vmax )
        Vmax = RootNode->value;

    /* CN() computes the conspiracy number of a node */
    if ( CN( INCREASE, RootNode, Vmax, MAX ) >= CT )
        do
            Vmax = Vmax - 1;
        while ( CN( INCREASE, RootNode, Vmax, MAX ) >= CT );

    else {
        do
            Vmax = Vmax + 1;
        while ( ( Vmax <= UB ) and
```

```
            ( CN( INCREASE, RootNode, Vmax, MAX ) < CT ) );

                                /* CN(Vmax) is >= CT    */
        Vmax = Vmax - 1;        /* so decrement it      */
    }

    return( Vmax );
}

/*-------------------------------------------------------------------------*/
/* Computes a new value for Vmin. Takes the old Vmin and the conspiracy */
/* threshold as parameters.                                                */
/*-------------------------------------------------------------------------*/
Compute_Vmin( Vmin, CT )
{
    /* similar to Compute_Vmax() */
}

/*-------------------------------------------------------------------------*/
/* Update the minimax values along the search path.                        */
/*-------------------------------------------------------------------------*/
Update_minimax_values()
{
    /* starting from the expanded node */
    for ( each node along the search path )
        if ( node is a MAX type node ) {
            Find son with highest minimax value;
            node->value = maxValue;
        }
        else {   /* node is a MIN type node */
            Find son with lowest minimax value;
            node->value = minValue;
        }
}

/*-------------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the   */
/* minimal conspiracy set for increasing the root node to Vmax.         */
/*-------------------------------------------------------------------------*/
struct Node *Rule_out_Vmax( Vmax )
{

    current_node = RootNode;
    PathDepth = 0;                  /* add RootNode to search path     */
    SearchPath[ PathDepth ] = current_node;
    node_type = MAX;                /* RootNode is a MAX type node     */

    while ( current_node is not a leaf node ) {
        if ( node_type == MAX ) {
            /* determine left-most son with minimum CN      */
            minSon = first son;
            min = INFINITY;
            for ( each son ) {
```

```
                temp = CN( INCREASE, son, Vmax, MIN );
                if ( temp < min ) {
                    min = temp;
                    minSon = son;
                }
            }

            /* add this son to the search path          */
            /* and make it the current_node             */
            current_node = minSon;
            PathDepth = PathDepth + 1;
            SearchPath[ PathDepth ] = current_node;
            /* current_node is now a MIN type node      */
            node_type = MIN;
        }
        else {                          /* node_type == MIN */
            for ( each son ) {
                /* find first son whose CN > 0          */
                if ( CN( INCREASE, son, Vmax, MAX ) > 0 ) {
                    /* add this son to search path      */
                    /* and make it the current_node     */
                    current_node = son;
                    PathDepth = PathDepth + 1;
                    SearchPath[ PathDepth ] = current_node;
                    /* current_node is a MAX node       */
                    node_type = MAX;
                    break;
                }
            }
        }
    }
    /* current_node is the leaf node chosen for expansion          */
    return( current_node );
}

/*-------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the  */
/* minimal conspiracy set for decreasing the root node to Vmin.        */
/*-------------------------------------------------------------------*/
struct Node *Rule_out_Vmin( Vmin )
{
    /* Similar to Rule_out_Vmax(). The rules for deciding which    */
    /* branch to descend down at MAX and MIN nodes are reversed.   */
}

/*-------------------------------------------------------------------*/
/* Expand a leaf node. Takes a pointer to the leaf node and the depth  */
/* of the leaf node as parameters.                                     */
/*-------------------------------------------------------------------*/
Expand( aNode, depth )
{
    Generate-the-successor-nodes();
    Evaluate-each-node();
```

```
    /* sort the sons appropriately  */
    if ( depth & 1 )  /* MIN node was expanded*/
        sort-in-ascending-order();
    else              /* MAX node was expanded*/
        sort-in-descending-order();
}

/*-------------------------------------------------------------------------*/
/* Compute a conspiracy number at a node. The parameters are:              */
/*      aNode     : a pointer to the node                                  */
/*      value     : the value we want to change the node to                */
/*      direction : the direction the change is to go ( either INCREASE */
/*                  or DECREASE ).                                       */
/*      node_type : the type of this node ( either MIN or MAX )          */
/*-------------------------------------------------------------------------*/
CN( direction, aNode, value, node_type )
{
    if ( aNode is a leaf node ) {
        /* apply formulas for leaf nodes    */
    }

    /* else aNode is an interior node so
       apply one of the formulas for computing
       the CN's. Note that there are recursive
       calls to CN() that occur here.        */

    else if ( ( node_type == MAX ) and
          ( direction == INCREASE ) ) {
        ....
    }
    else if ( ( node_type == MAX ) and
          ( direction == DECREASE ) ) {
        ....
    }
    else if ( ( node_type == MIN ) and
          ( direction == INCREASE ) ) {
        ....
    }
    else { /* ( node_type == MIN ) and
              ( direction == DECREASE )     */
        ....
    }
}
```

Appendix 2

The Storage Intensive Implementation Version

The following is a pseudo-code description of the Storage Intensive Implementation version of the Conspiracy Numbers algorithm. The main feature of this version is that it stores conspiracy numbers for all the legal node values at all the interior nodes of the tree and therefore does not have to traverse the entire tree in order to compute a conspiracy number. However it has to update them at nodes along the search path each time a node is expanded.

```
/*-------------------------------------------------------------------------*/
/* Some constants used by the program                                      */
/*-------------------------------------------------------------------------*/

#define LB  -40                         /* lower bound on node values   */
#define UB  40                          /* upper bound on node values   */

#define MAX   0                         /* for indicating type of node  */
#define MIN   1

#define DECREASE   0                    /* for purpose of computing CN's*/
#define INCREASE   1                    /* indicates direction of change*/

#define INFINITY   255                  /* maximum conspiracy number    */

/*-------------------------------------------------------------------------*/
/* Info stored at all nodes of the tree                                    */
/*-------------------------------------------------------------------------*/

struct Node {
        short int value;                /* minimax value of this node   */
        struct Extension *part2;        /* pointer to second part       */
};

/* Additional info stored at all interior nodes                           */

struct Extension {
        short int numSons;              /* number of sons               */
        unsigned char CN[ UB-LB+1 ];    /* conspiracy numbers           */
        struct Node *sons;              /* pointer to array of sons     */
};

/*-------------------------------------------------------------------------*/
```

```
/* Structure for storing a path between two nodes                    */
/*-------------------------------------------------------------------*/

struct Node *SearchPath[ MAXDEPTH ];
int PathDepth;                          /* index of last node in path  */

struct Node *RootNode;                  /* pointer to root node        */

/*-------------------------------------------------------------------*/
/* Main routine for the Conspiracy Numbers Algorithm. Takes a        */
/* conspiracy threshold as parameter.                                */
/*-------------------------------------------------------------------*/
Top_Level( CT )
{
        /* Identical to Top_Level() in Basic Implementation except that
           there is a call to the new routine Update_CN() that is made
           after the call to Update_minimax_values()                  */
}

/*-------------------------------------------------------------------*/
/* Computes a new value for Vmax. Takes the old Vmax and the conspiracy */
/* threshold as parameters.                                          */
/*-------------------------------------------------------------------*/
Compute_Vmax( Vmax, CT )
{
        /* Identical to Compute_Vmax() in Basic Implementation       */
}

/*-------------------------------------------------------------------*/
/* Computes a new value for Vmin. Takes the old Vmin and the conspiracy */
/* threshold as parameters.                                          */
/*-------------------------------------------------------------------*/
Compute_Vmin( Vmin, CT )
{
        /* Identical to Compute_Vmin() in Basic Implementation       */
}

/*-------------------------------------------------------------------*/
/* Update the minimax values along the search path.                  */
/*-------------------------------------------------------------------*/
Update_minimax_values()
{
        /* Identical to Update_minimax_values() in Basic Implementation */
}

/*-------------------------------------------------------------------*/
/* Update the conspiracy numbers along the search path.              */
/*-------------------------------------------------------------------*/
Update_CN()
{
    /* starting from the expanded node  */
    for ( each node along the search path ) {
```

```
            if ( node is a MAX type node )
                node_type = MAX;
            else
                node_type = MIN;

            for ( each value v from LB to UB ) {

                if( v < node->value )
                    direction = DECREASE;
                else
                    direction = INCREASE;

                /* apply one of the formulas for computing
                   a CN at an interior node.                */

                if ( ( node_type == MAX ) and
                     ( direction == INCREASE ) ) {
                    ....
                }

                else if ( ( node_type == MAX ) and
                       ( direction == DECREASE ) ) {
                        ....
                }

                else if ( ( node_type == MIN ) and
                       ( direction == INCREASE ) ) {
                        ....
                }

                else /* ( node_type == MIN ) and  */
                     /* ( direction == DECREASE ) */ {
                        ....
                }

                /* store result in node.                */
                node->part2->CN[ v - LB ] = answer;
            }
        }
    }
```

```
/*-------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the  */
/* minimal conspiracy set for increasing the root node to Vmax.        */
/*-------------------------------------------------------------------*/
struct Node *Rule_out_Vmax( Vmax )
{
    current_node = RootNode;
    PathDepth = 0;                  /* add RootNode to search path  */
    SearchPath[ PathDepth ] = current_node;
    node_type = MAX;                /* RootNode is a MAX type node  */
```

```
    while ( current_node is not a leaf node ) {
        if ( node_type == MAX ) {
            CN_Vmax = CN( INCREASE, current_node, Vmax, MAX );

            /* find first son whose CN is equal to CN_Vmax  */
            /* this is also the son with minimum CN         */
            for ( each son )
                if ( CN( INCREASE, son, Vmax, MIN ) == CN_Vmax){
                    /* add this son to search path  */
                    /* and make it the current_node */
                    current_node = son;
                    PathDepth = PathDepth + 1;
                    SearchPath[ PathDepth ] = current_node;
                    /* current_node is a MIN node   */
                    node_type = MIN;
                    break;
                }
        }
        else {                  /* node_type == MIN */
            for ( each son )
                /* find first son whose CN > 0      */
                if ( CN( INCREASE, son, Vmax, MAX ) > 0 ) {
                    /* add this son to search path  */
                    /* and make it the current_node */
                    current_node = son;
                    PathDepth = PathDepth + 1;
                    SearchPath[ PathDepth ] = current_node;
                    /* current_node is a MAX node   */
                    node_type = MAX;
                    break;
                }
        }
    }
    /* current_node is the leaf node chosen for expansion              */
    return( current_node );
}


/*-------------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the      */
/* minimal conspiracy set for decreasing the root node to Vmin.            */
/*-------------------------------------------------------------------------*/
struct Node *Rule_out_Vmin( Vmin )
{
        /* Similar to Rule_out_Vmax(). The rules for deciding which        */
        /* branch to descend down at MAX and MIN nodes are reversed.       */
}

/*-------------------------------------------------------------------------*/
/* Expand a leaf node. Takes a pointer to the leaf node and the depth      */
/* of the leaf node as parameters.                                         */
/*-------------------------------------------------------------------------*/
Expand( aNode, depth )
```

```
{
        /* Identical to Expand() in Basic Implementation                */
}

/*-------------------------------------------------------------------------*/
/* Get a conspiracy number from a node. The parameters are:                */
/*      aNode      : a pointer to the node                                 */
/*      value      : the value we want to change the node to               */
/*      direction  : the direction the change is to go ( either INCREASE */
/*                   or DECREASE ).                                       */
/*      node_type  : the type of this node ( either MIN or MAX )          */
/*-------------------------------------------------------------------------*/
CN( direction, aNode, value, node_type )
{

    if ( aNode is a leaf node ) {
        /* apply formulas for leaf nodes        */
    }

    /* else aNode is an interior node so retrieve
       the conspiracy number from its structure  */

    else if ( ( node_type == MAX ) and
              ( direction == INCREASE ) ) {
        if ( aNode->value >= value )
            return( 0 );
        else
            return( aNode->part2->CN[ value-LB ] );
    }
    else if ( ( node_type == MAX ) and
              ( direction == DECREASE ) ) {
        ....
    }
    else if ( ( node_type == MIN ) and
              ( direction == INCREASE ) ) {
        ....
    }
    else { /* ( node_type == MIN ) and  */
           /* ( direction == DECREASE ) */
        ....
    }
}
```

Appendix 3

The Compromise Implementation Version

The following is a pseudo-code description of the Compromise Implementation version of the Conspiracy Numbers algorithm. The main feature of this version is that it stores conspiracy numbers for 4 node values at all the nodes of the tree.

```
/*-------------------------------------------------------------------------*/
/* Some constants used by the program                                      */
/*-------------------------------------------------------------------------*/

#define LB  -40                         /* lower bound on node values   */
#define UB  40                          /* upper bound on node values   */

#define MAX   0                         /* for indicating type of node  */
#define MIN   1

#define DECREASE   0                    /* for purpose of computing CN's*/
#define INCREASE   1                    /* indicates direction of change*/

#define INFINITY   255                  /* maximum conspiracy number    */

/*-------------------------------------------------------------------------*/
/* Info stored at all nodes of the tree                                    */
/*-------------------------------------------------------------------------*/

struct Node {
        short int value;                /* minimax value of this node   */
        short int numSons;              /* number of sons               */
        struct Node *sons;              /* pointer to array of sons     */
        unsigned char CNvmin, CNvmin1;  /* CNs for Vmin and Vmin - 1    */
        unsigned char CNvmax, CNvmax1;  /* CNs for Vmax and Vmax - 1    */
};

/*-------------------------------------------------------------------------*/
/* Structure for storing a path between two nodes                          */
/*-------------------------------------------------------------------------*/

struct Node *SearchPath[ MAXDEPTH ];
int PathDepth;                          /* index of last node in path   */

struct Node *RootNode;                  /* pointer to root node         */

/*-------------------------------------------------------------------------*/
/* Vmax, Vmin, Vroot should be global in this version                      */
/*-------------------------------------------------------------------------*/
```

```
int Vmax, Vmin, Vroot;

/*-------------------------------------------------------------------------*/
/* Main routine for the Conspiracy Numbers Algorithm. Takes a              */
/* conspiracy threshold as parameter.                                      */
/*-------------------------------------------------------------------------*/
Top_Level( CT )
{
    if ( RootNode == NULL ) {
        Initialize_root();
        Vmin = Vmax = Vroot = RootNode->value;
        RootNode->CNvmin = RootNode->CNvmin1 = 0;
        RootNode->CNvmax = RootNode->CNvmax1 = 0;
    }

    Vroot = RootNode->value;
    Vmin = Compute_Vmin( Vmin, CT );
    Vmax = Compute_Vmax( Vmax, CT );

    while ( Vmin != Vmax ) {
        /* Identical to the while loop of the
           Top_Level() routine in the Storage
           Intensive Version.                    */
    }
    return( Vroot );
}

/*-------------------------------------------------------------------------*/
/* Computes a new value for Vmax. Takes the old Vmax and the conspiracy */
/* threshold as parameters.                                                */
/*-------------------------------------------------------------------------*/
Compute_Vmax( Vmax, CT )
{
    if ( RootNode->value > Vmax )
        Vmax = RootNode->value;

    if ( CN( INCREASE, RootNode, Vmax, MAX ) >= CT )
        do
            Vmax = Vmax - 1;
        while ( CN_Vmax( RootNode, Vmax, MAX ) >= CT );

    else if ( CN( INCREASE, RootNode, Vmax+1, MAX ) < CT )
        do
            Vmax = Vmax + 1;
        while ( CN_Vmax1( RootNode, Vmax+1, MAX ) < CT);


    return( Vmax );
}

/*-------------------------------------------------------------------------*/
/* Computes a new value for Vmin. Takes the old Vmin and the conspiracy */
/* threshold as parameters.                                                */
/*                                                                         */
```

```
/*-------------------------------------------------------------------------*/
Compute_Vmin( Vmin, CT )
{
    /* Similar to Compute_Vmax() */
}

/*-------------------------------------------------------------------------*/
/* Compute the conspiracy number corresponding to Vmax at aNode and at  */
/* all relevant nodes below it. Here it is assumed that the new Vmax is */
/* equal to "old_Vmax - 1" and the new Vmax1 is equal to old_Vmax.       */
/* Therefore, to avoid recomputing conspiracy numbers for the new Vmax1,*/
/* the old conspiracy number for Vmax is simply copied to CNvmax1.       */
/*-------------------------------------------------------------------------*/
CN_Vmax( aNode, Vmax, node_type )
{
    if ( aNode is a leaf node ) {
        /* apply CN formulas for leaf nodes */
    }

    else if ( node_type == MAX ) {
        if ( aNode->value == Vmax ) {
            answer = 0;
            /* relevant nodes wrt Vmax1 must be visited    */
            copy_CNvmax( aNode, Vmax );
        }
        else {
            /* apply CN formula for increasing value to Vmax
               at a MAX interior node. Here the function
               must be recursively called for each son.    */
        }
    }

    else /* ( node_type == MIN ) */ {
        if ( aNode->value == Vmax ) {
            answer = 0;
            /* relevant nodes wrt Vmax1 must be visited    */
            copy_CNvmax( aNode, Vmax );
        }
        else {
            /* apply CN formula for increasing value to Vmax
               at a MIN interior node. Here the function
               must be recursively called for each son.    */
        }
    }

    /* the old CNvmax becomes the new CNvmax1 */
    aNode->CNvmax1 = aNode->CNvmax;
    aNode->CNvmax = answer;
    return( answer );
}

/*-------------------------------------------------------------------------*/
/* Copy the old value for CNvmax to CNvmax1 at all relevant nodes below */
```

```
/* "node". These nodes are not visited by the routine CN_Vmax().        */
/*-------------------------------------------------------------------------*/
copy_CNvmax( node, Vmax )
{
    for ( each son ) {
        son->CNvmaxi = son->CNvmax;
        if ( son's value <= Vmax )
            copy_CNvmax( son, Vmax );
    }
}

/*-------------------------------------------------------------------------*/
/* Compute the conspiracy number corresponding to Vmin at aNode and at  */
/* all relevant nodes below it. Also, copies the conspiracy number for  */
/* the old Vmin to CNvmini.                                             */
/*-------------------------------------------------------------------------*/
CN_Vmin( aNode, Vmin, node_type )
{
    /* Similar to CN_Vmax(). */
}

/*-------------------------------------------------------------------------*/
/* Compute the conspiracy number corresponding to Vmaxi at aNode and at */
/* all relevant nodes below it. Also, copies the conspiracy number for  */
/* the old Vmaxi to CNvmax.                                             */
/*-------------------------------------------------------------------------*/
CN_Vmaxi( aNode, Vmaxi, node_type )
{
    /* Similar to CN_Vmax() except that it does not call
       a function similar to copy_CNvmax because when it
       stops traversing the tree at a node, there are no
       relevant nodes wrt Vmax or Vmaxi below it.       */
}

/*-------------------------------------------------------------------------*/
/* Compute the conspiracy number corresponding to Vmini at aNode and at */
/* all relevant nodes below it. Also, copies the conspiracy number for  */
/* the old Vmini to CNvmin.                                             */
/*-------------------------------------------------------------------------*/
CN_Vmini( aNode, Vmini, node_type )
{
    /* Similar to CN_Vmax() except that it does not call
       a function similar to copy_CNvmax because when it
       stops traversing the tree at a node, there are no
       relevant nodes wrt Vmin or Vmini below it.       */
}

/*-------------------------------------------------------------------------*/
/* Update the minimax values along the search path.                     */
/*-------------------------------------------------------------------------*/
Update_minimax_values()
{
    /* Identical to Update_minimax_values() in Basic Implementation */
```

```
}

/*-------------------------------------------------------------------*/
/* Update the conspiracy numbers along the search path.              */
/*-------------------------------------------------------------------*/
Update_CN()
{
    /* first check for "new relevant" nodes
       and fix their conspiracy numbers    */
    if ( the Rule_out_Vmax strategy was used ) {
        fixup_CNvmin( Vmin );
        fixup_CNvmin1( Vmin - 1 );
    }
    else {
        fixup_CNvmax( Vmax );
        fixup_CNvmax1( Vmax + 1 );
    }

    /* now update the conspiracy numbers
       corresponding to Vmax, Vmax + 1,
       Vmin, and Vmin - 1.              */
    Update_CNvmax1( Vmax + 1 );
    Update_CNvmax( Vmax );
    Update_CNvmin( Vmin );
    Update_CNvmin1( Vmin - 1 );

}

/*-------------------------------------------------------------------*/
/* Check if new relevant nodes with respect to Vmin have occurred as a  */
/* result of a leaf node being expanded. If there are any, make sure    */
/* that they have their conspiracy number corresponding to Vmin         */
/* recomputed.                                                          */
/*-------------------------------------------------------------------*/
fixup_CNvmin( Vmin )
{
    /* starting at the root node */
    for ( each node along the search path ) {
        if ( the node's old CNvmin == 0 ) {
            if ( node's new value > Vmin )
                recompute_CNvmin( node, Vmin, node_type );
            break;
        }
    }
}

/*-------------------------------------------------------------------*/
/* Recompute the conspiracy numbers corresponding to Vmin at each       */
/* relevant node below the parameter node "aNode". Other parameters are */
/*      Vmin      : the current value for Vmin                          */
/*      node_type : the type of node aNode is ( either MIN or MAX )     */
/*-------------------------------------------------------------------*/
recompute_CNvmin( aNode, Vmin, node_type )
```

```
{
    if ( aNode is a leaf node ) {
        /* apply CN formulas for leaf nodes */
    }

    else if ( node_type == MAX ) {
        /* apply CN formula for decreasing value to Vmin
           at a MAX interior node. Here the function
           must be recursively called for each son.    */
    }

    else /* ( node_type == MIN ) */ {
        /* apply CN formula for decreasing value to Vmin
           at a MIN interior node. Here the function
           must be recursively called for each son.    */
    }

    aNode->CNvmin = answer;
    return( answer );
}

/*-------------------------------------------------------------------------*/
/* These routines check if new relevant nodes with respect to Vmin - 1,    */
/* Vmax, Vmax + 1 have occurred as a result of a leaf node being           */
/* expanded. If there are any, they make sure that these nodes have        */
/* their conspiracy number recomputed.                                     */
/*-------------------------------------------------------------------------*/
fixup_CNvmin1( Vmin1 )
fixup_CNvmax( Vmax )
fixup_CNvmax1( Vmax1 )
{
    /* Similar to fixup_CNvmin() */
}

/*-------------------------------------------------------------------------*/
/* These routines recompute the conspiracy numbers corresponding to        */
/* Vmin - 1, Vmax, and Vmax + 1 at each relevant node below the            */
/* parameter node "aNode".                                                 */
/*-------------------------------------------------------------------------*/
recompute_CNvmin1( aNode, Vmin1, node_type )
recompute_CNvmax( aNode, Vmax, node_type )
recompute_CNvmax1( aNode, Vmax1, node_type )
{
    /* Similar to recompute_CNvmin() */
}

/*-------------------------------------------------------------------------*/
/* Update the conspiracy numbers corresponding to Vmax at each new leaf    */
/* node and at each node along the current search path.                     */
/*-------------------------------------------------------------------------*/
Update_CNvmax( Vmax )
{
    for ( each new leaf node ) {
```

```
            /* apply CN formula for increasing value
               to Vmax at a leaf node                          */
        }

        /* starting from the expanded node */
        for ( each node along the search path ) {

            if ( node is of type MAX ) {
                /* apply CN formula for increasing value
                   to Vmax at a MAX interior node        */
            }

            else /* node is of type MIN */ {
                /* apply CN formula for increasing value
                   to Vmax at a MIN interior node        */
            }

            node->CNvmax = answer;
        }
    }

/*-------------------------------------------------------------------------*/
/* These routines update the conspiracy numbers corresponding to           */
/* Vmax + 1, Vmin, and Vmin - 1 at each new leaf node and at each node     */
/* along the current search path.                                          */
/*-------------------------------------------------------------------------*/
Update_CNvmax1( Vmax1 )
Update_CNvmin( Vmin )
Update_CNvmin1( Vmin1 )
{
    /* Similar to Update_CNVmax() */
}

/*-------------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the      */
/* minimal conspiracy set for increasing the root node to Vmax.            */
/*-------------------------------------------------------------------------*/
struct Node *Rule_out_Vmax( Vmax )
{
       /* Identical to Rule_out_Vmax() in Storage Intensive Version    */
}

/*-------------------------------------------------------------------------*/
/* Finds a leaf node for expansion. This leaf node is a member of the      */
/* minimal conspiracy set for decreasing the root node to Vmin.            */
/*-------------------------------------------------------------------------*/
struct Node *Rule_out_Vmin( Vmin )
{
       /* Identical to Rule_out_Vmin() in Storage Intensive Version    */
}

/*-------------------------------------------------------------------------*/
/* Expand a leaf node. Takes a pointer to the leaf node and the depth      */
```

```
/* of the leaf node as parameters.                                          */
/*--------------------------------------------------------------------------*/
Expand( aNode, depth )
{
        /* Identical to Expand() in Storage Intensive Version               */
}

/*--------------------------------------------------------------------------*/
/* Retrieve a conspiracy number from a node. The parameters are:            */
/* \     aNode      : a pointer to the node                                 */
/*       value      : the value we want to change the node to. The          */
/*                    program never calls this routine for any value        */
/*                    other than Vmin, Vmin - 1, Vmax, Vmax + 1.            */
/*       direction  : the direction the change is to go ( either INCREASE */
/*                    or DECREASE ).                                        */
/*       node_type  : the type of this node ( either MIN or MAX )         */
/*--------------------------------------------------------------------------*/
CN( direction, aNode, value, node_type )
{
    if ( ( value == Vmin - 1 ) and ( direction == DECREASE ) )
        return( aNode->CNvmini );

    else if ( ( value == Vmin ) and ( direction == DECREASE ) )
        return( aNode->CNvmin );

    else if ( ( value == Vmax ) and ( direction == INCREASE ) )
        return( aNode->CNvmax );

    else   /* ( value == Vmax + 1 ) and ( direction == INCREASE )   */
        return( aNode->CNvmaxi );
}
```